Department of Informatics and Mathematics
Chair of Distributed Information Systems

Doctoral Thesis

Efficient Exchange and Processing of Semi-structured Data in the Embedded Domain

Daniel Peintner

December 2014

Abstract

The Internet is a global system of interconnected computers and computer networks where semi-structured data has been successfully applied for exchanging information. In nowadays Internet the huge range of actors, the large diversity of the associated device classes and domains, and the enormous amount of resource-restricted controllers in this system created new requirements and coined also a new term. *Internet of Things* (IoT), in this regard, refers to identifiable objects (things) and their virtual representations in an Internet-like structure. The fundamental question the thesis tries to answer is whether and how the same semi-structured data can be also applied to the IoT and the embedded domain in spite of resource-limited controllers.

In order to discuss this question properties and requirements of embedded networks with regard to the IoT domain have been collected and evaluated. Thereafter the omnipresent semi-structured data exchange format in the Web, the Extensible Markup Language (XML), has been validated. The result was a list of missing requirements such as a compact representation, a representation that can be generated and consumed fast and also allows a small footprint implementation. To address the compiled requirements a binary representation of XML which nowadays is known as W3Cs Efficient XML Interchange (EXI) format has been accomplished which simultaneously optimizes performance and the utilization of computational resources and is designed to be compatible with XML. Moreover, in this work the format has been practically validated and tested. Addressing the needs of the embedded domain one result of this analyzes were optimizations to constrain runtime memory usage and to predict memory growth at runtime. A concept introduced in this thesis is LazyDOM which reduces memory requirements when processing and querying data. By means of a newly proposed code generation technique processing of EXI on ultra-constrained device classes has been enabled and resulting format modifications have been adopted by the W3C standardization.

The research work described in this thesis on efficiently exchanging and processing semi-structured data on constrained embedded devices has not only triggered modifications in the W3C EXI format but even is already adopted in domain specific application standards and implementations. The above mentioned optimizations such as predictably limit the memory growth at runtime have been contributed, discussed and evaluated by the W3C experts and become a core part of the EXI specification. Even more significantly from the IoT perspective these optimizations provide the basis for the adoption of this technology in ISO and IEC standardization which is the first time for automotive and power industry to use IoT in the control plane. The implementation of EXI to conduct the evaluation as part of this thesis has become the de-facto open source reference implementation of EXI and became the basis of a number of other reference implementations such as the OpenV2G project that provides the reference implementation of the communication interface in ISO/IEC 15118.

In summary the conducted research work has evaluated the options to adapt semi-structured data for the constrained embedded domain, proposed modifications and evaluated those under realistic conditions. This made it relevant for the technology as well as for application standardization despite the short period of this work. As such the research can now be taken as a basis for further challenges in the IoT field namely adopting concepts of the Semantic Web and adapting those to stimulate the quickly expanding eco-system of embedded devices.

Dedicated to my beloved family

Acknowledgements

I am deeply grateful to my advisors Prof. Dr. Harald Kosch and Dr. Jörg Heuer. Both continuously supported me during the process of working on my doctoral thesis. I would like to say thanks, not only for guiding and helping me through this journey, but also for the personal support in various aspects. Thank you, without your help and motivation this dissertation would not have been possible. I would also like to thank Prof. Dr. Hermann Hellwagner for his willingness to peer-review this dissertation.

I thank all my colleagues at the Siemens AG research & development laboratory that accompanied me with enjoyable and fruitful discussions. I would like to mention Daniel Vogelheim, Richards Kuntschke, Thomas Kurz, Andreas Scholz, Francesc Sanahuja, Joachim Laier, and Andreas Ziller. I really appreciate spending time with you, chatting about various topics that frequently turned out to be beneficial for the actual thesis work as well. My special thanks go to Sebastian Käbisch and Peter Amon that supported the thesis writing with their technical and personal feedback throughout the entire developing process. Moreover, Sebastian and Peter helped also to iron out remaining issues to make the work consistent and harmonious.

Accompanying my research, I got the chance to be part of an international standardization working group that allowed me to get to know very inspiring people all over the world. I really appreciated any experience I was able to make. Special thanks go to Takuki Kamiya, Youenn Fablet, and John Schneider that helped me widen my view with their expertise and general knowledge but also with their openness and amity.

Last but not least, I am very thankful to my family, which supported and motivated me over the last couple of years with long-lasting patience, love, and understanding.

Contents

Contents i

List of Figures v

List of Tables vii

I Introduction 1

1 Motivation 3

2 Requirements of the Embedded Domain 5

3 Semi-structured Data in the Embedded Domain 7

4 Goal and Structure 9

II Overview of Semi-structured Data in the Embedded Domain 13

5 XML Technologies 15

5.1 XML Fundamentals 16

5.1.1 XML and XML Schema 16

5.1.2 XML Optimization Fields 19

5.2 XML Content Density 20

5.3 XML Test Data Characteristics 21

6 Binary XML 25

6.1 Related Work 25

6.1.1 XMill 27

6.1.2 WBXML 28

6.1.3 ASN.1 28
6.1.4 FastInfoset 28
6.1.5 BiM 29
6.2 Efficient XML Interchange (EXI) Format 30
6.2.1 Basic Concepts 31
6.2.2 EXI Grammars 33
6.2.3 EXI String Table 38
6.2.4 Variable-length Unsigned Integer Coding 41
6.2.5 EXI Compression 41
6.2.6 Summary 43
6.3 EXI Metrics 43
6.3.1 EXI Content Density 43
6.3.2 EXI Efficiency 45
6.4 Discussion 45

III Optimization of XML Technologies **49**

7 Efficient Feature-complete EXI Processor **51**
7.1 Implementation Techniques 52
7.1.1 Element Context Stack 54
7.1.2 Qualified Name Context 55
7.1.3 Grammars 57
7.1.4 String Table 58
7.2 Results and Discussion 60
7.2.1 Compression 61
7.2.2 Processing Efficiency 65
7.2.3 Optimization Outlook 69

8 Optimized Datatype Representation **71**
8.1 Application Area 72
8.1.1 Scalable Vector Graphics 74
8.1.2 Applying EXI to Rich Web Applications 75
8.2 Formal Grammar 77
8.2.1 Chomsky Hierarchy 78
8.2.2 Backus-Naur Form 78
8.2.3 Parser Observations 80
8.2.4 XML Schema and EXI Grammars 80
8.3 Concept of an Optimized Datatype Representation 81
8.3.1 Datatype Representation Generation 81
8.3.2 BNF Extraction Process 82
8.3.3 BNF-based Datatype Representation Approach 85

8.4 Results and Discussion 86
8.4.1 Measurements and Evaluation 86
8.4.2 Entropy Coding 87

9 EXI Grammar Representation 89
9.1 XML Schema Exchange 89
9.1.1 Requirements 90
9.1.2 XML Schema Knowledge Exchange for Binary XML 91
9.1.3 XML Schema Knowledge Exchange for EXI 92
9.2 Contents of XML Schema Exchange 92
9.2.1 Qualified Names 92
9.2.2 EXI Grammars 93
9.2.3 XML Schema Exchange Representation for EXI 99
9.2.4 Schema for Grammars 99
9.3 Results and Discussion 104
9.3.1 EXI Grammars Start-up Costs 104
9.3.2 Test Set 107
9.3.3 Compression Measurements 108
9.3.4 Parsing Time Measurements 110
9.3.5 Summary 112

10 Demand-tailored EXI Processor 113
10.1 EXI Grammar Transformations 114
10.1.1 Properties of EXI Grammars 115
10.1.2 Mapping EXI Grammars to Finite State Machines 115
10.1.3 Mapping EXI Grammars to Source Code 116
10.2 Automatic EXI Processor Generation 119
10.2.1 Automatic Source Code Generation according to EXI Grammars 119
10.2.2 Application Programming Interface 120
10.2.3 EXI Decoder Generation 122
10.2.4 Automatic Databinding 130
10.3 Results and Discussion 131
10.3.1 Test Candidates 131
10.3.2 Code Footprint Numbers 133
10.3.3 Parser Performance 136
10.3.4 Usability Evaluation and Outlook 136

11 Memory-Sensitive XML Querying 139
11.1 Memory-Sensitive Model 140
11.1.1 LazyDOM Concept 140
11.1.2 LazyDOM Requirements 141

11.2 Technical Realization . . . 141
11.2.1 Indexing Mechanism . . . 142
11.2.2 LazyDOM Applicability . . . 143
11.3 Results and Discussion . . . 144
11.3.1 Test Data . . . 144
11.3.2 Query Set . . . 145
11.3.3 Performance Measurements . . . 145
11.3.4 Design Guidelines . . . 148
11.3.5 Related Work . . . 149

IV Résumé **151**

12 Contributions **153**

13 Conclusions **157**

V Appendices **159**

Bibliography **161**

List of Symbols and Abbreviations **169**

Verbose Listings **173**

List of Figures

4.1 Smart Grid and Smart Energy 9

5.1 XML Content Density 22

6.1 Processing steps when compressing XML 26
6.2 EXI Body stream for Listing 5.1 33
6.3 Built-in element grammar for SE(note) 34
6.4 Strict schema-informed grammar for StartElement(note) 35
6.5 Grammar Event Code Tree for grammar state *StartTag1* in Figure 6.4 36
6.6 String Table - Entries in uri, prefix, and local-name Partitions 40
6.7 String Table - (Final) Entries in Value Partition 41
6.8 EXI Compression 42

7.1 EXI Overview - Fully conforming EXI Processor 53
7.2 EXIficient ElementContext 55
7.3 EXIficient QNameContext 56
7.4 EXIficient Grammars 58
7.5 String Table Implementation Details – Entries in Value Partition . . . 59
7.6 String Table Implementation Details – Optimized Value Partition . . . 60
7.7 EXI Compression 61
7.8 EXI Compression - High XML Content Density 62
7.9 EXI Compression - Low XML Content Density - Large documents . . 63
7.10 EXI Compression - Low XML Content Density - Small documents . . 64
7.11 EXI Compression - Low XML Content Density - Tiny documents . . . 64
7.12 EXI Processing Time - Encoding 66
7.13 EXI Processing Time - Decoding 67
7.14 EXI Decoding - High XML Content Density 68
7.15 EXI Decoding - Low XML Content Density - Large documents 68
7.16 EXI Decoding - Low XML Content Density - Small documents 69
7.17 EXI Decoding - Low XML Content Density - Tiny documents 70

8.1 Hierarchy of XML Schema Datatypes [67] 72
8.2 Example of an SVG path 74
8.3 EXI Compression of SVG Data 75
8.4 SVG Data Distribution for XML and EXI Representation 77
8.5 EXI Representation Idea 82
8.6 SVG Data Distribution by means of the EXI-BNF-aware solution . . . 87

9.1 Schema-informed EXI grammars for notebook.xsd 93
9.2 Schema-informed EXI grammars for modified notebook.xsd 98
9.3 Schema for Grammars - Root element `exiGrammar` 100
9.4 Schema for Grammars - Element `qnames` 101
9.5 Schema for Grammars - Element `qnameContext` 102
9.6 Schema for Grammars - Element `grammars` 103
9.7 Schema for Grammars - Type Production 104
9.8 Compression Results for XML Schema Exchange Formats 109
9.9 Compression Results for XML Schema Exchange Formats with DEFLATE 111
9.10 EXI Grammars Parsing Time 112

10.1 EXI Grammar for SOAP element `Envelope` content model 114
10.2 Schema-informed EXI grammars for notebook.xsd 117
10.3 Schema-informed EXI grammars for notebook.xsd with grammar IDs . 120
10.4 XML/EXI Parser Performance 137

11.1 Lazy Document Object Model containing loaded and not loaded elements 140
11.2 XML Schema for index structure 142
11.3 Memory consumption and query execution times 146

List of Tables

5.1 Test-groups classified by number of XML documents per cluster 24

6.1 EXI events 32
6.2 *Naive* Event Code Assignment vs. EXI Event Code Assignment . . . 37
6.3 Fidelity options 38
6.4 Requirements of binary XML formats 47

8.1 Built-in EXI datatype representations 73
8.2 EXI Datatype Efficiency Ratio for SVG Test Documents 76

9.1 Schema-informed EXI document grammars 94
9.2 Schema-informed EXI fragment grammars 95
9.3 Schema-informed EXI StartTagContent and ElementContent grammars 97

10.1 XML/EXI parser library size (in Bytes) 133
10.2 XML parser requirements (size in Bytes) 135
10.3 EXI parser requirements based on the used schema information (size in Bytes) 136
10.4 Number of EXI grammar states and events 136

11.1 XMark test documents 144

Part I

Introduction

Chapter 1

Motivation

The Internet is a global system of interconnected computers and computer networks that use standardized Internet protocols. In the past, typically powerful servers have been delivering web content to clients such as desktop computers. In recent years, the trend continued towards more devices, and especially new device classes, which are becoming part of the overall Internet-based infrastructure. Also, more and more very diverse areas emerge to interact in one way or the other. The term *Internet of Things* (IoT), presumably first coined by Kevin Ashton [53] back in 1999, is often used to describe this phenomenon of the future Internet of multiple diverse participants. The term was influenced by the work of the Auto-ID Center at the Massachusetts Institute of Technology (MIT), which in 1999 started to design and propagate a cross-company RFID infrastructure [71]. A very similar notion was used by Neil Gershenfeld, also from the MIT Media Lab, around the same time in his popular book "When Things Start to Think" [32]. In general, the term *Internet of Things* refers to identifiable objects (things) and their virtual representations in an Internet-like structure.

The goal of the IoT attempt is that all devices are fully integrated across all participants by offering a seamless communication. However, the information that are to be exchanged and/or communicated show a high diversity because of the large amount of domains and stakeholders, the relevant use cases, and the large number of properties required to support the use cases. This leads to a long list of desirable properties in which it is mandatory for the exchange format to be processable on a wide variety of platforms (space efficient) and easy to consume (processing efficient) [19, 18].

So far many domain specific interchange formats have been developed to make it possible that small and restricted devices such as a microcontroller in a refrigerator can place an order on a web shopping platform. The emerging number of diverse parties involved in exchanging information makes it necessary to elaborate

a more device-friendly protocol to support various units varying from powerful machines to very restricted devices in regard to processing power and memory consumption.

Another example are smart houses and homes, or in an even broader range smart grids, demanding an exchange format that is suitable for various device classes. The ZigBee Smart Energy 2.0 specification [7], which is still under development, defines an IP-based protocol to monitor, control, inform, and automate the delivery and use of energy and water. Moreover, also the ISO/IEC 15118 in the context of ISO TC22/SC3 [3, 4] under the keyword "Electrical and electronic equipment" is working on standardizing the communication between electrical and electronic equipment. As such, the DIN 70121 [2] standard provides a solid basis for Battery Electric Vehicles (BEV) or Plug-in Hybrid Electric Vehicles (PHEV) and the Electric Vehicle Supply Equipment (EVSE).

Summarizing all these ambitious efforts, it can be stated that there is need for an *Efficient Exchange and Processing of Semi-structured Data*, in particular in the *Embedded Domain* providing support for the most restrictive domains.

Chapter 2

Requirements of the Embedded Domain

An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints. It is usually *embedded* as a part of a complete device (e.g., domestic refrigerator or other home appliances). A general purpose computer, in turn, is designed to be flexible and to meet a wide range of end-user needs.

A *microcontroller* (sometimes abbreviated μC, uC, or MCU) is a small computer containing a processor, memory, and programmable input/output peripherals on a single integrated circuit. On the contrary, general-purpose microprocessors need to add RAM, ROM, I/O ports, and timers externally to make them functional. That said, the fixed amount of on-chip ROM, RAM, and number of input/output ports makes microcontrollers ideal for many applications in which cost per unit and packaging space are critical. However, there is the downside in regard to versatility [54].

Very often the terms *embedded processor* and *microcontroller* are used interchangeably. Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, medical devices, remote controls, office machines, appliances, cellular phones, TVs and cameras, toys, and other embedded systems. Typically, each microcontroller runs one application software only that is said to be *burned into ROM* [54].

Nowadays microcontrollers are cheap when bought in quantity. Further, according to Semico Research Corporation[1], more than 50% of all processors sold in the world are 8-bit microcontrollers and microprocessors. However, there are also 16-bit and 32-bit microcontrollers made by various chip makers. According

[1] Semico Website, `http://www.semico.com/`

to [54], there are a few relevant criteria for choosing the appropriate microcontroller depending on the requirements:

- Speed
- Packaging
- Cost per unit
- Power consumption
- Amount of RAM and ROM on chip
- The number of I/O pins and timers
- How easy to upgrade to higher performance or lower power-consumption versions

The list gives a very good idea about the concerns and the fundamental properties. The less resources an application or software accounts for, the cheaper, smaller, and less powerful a microcontroller can be built. Especially in regard to economic efficiency it plays a major role to select the adequate embedded processor that fits the needs. That said, no more resources than actually needed should be deployed given to the implication on cost and power consumption.

This is also the reason why most embedded systems are built using Assembly or C language. Compilers produce hex files that are flushed to the ROM of a microcontroller. Programming C is less time consuming, but has typically larger hex file size compared to Assembly. However, the reasons for writing programs in C are manifold. It is easier to write in C than Assembly and one can use code available in libraries. Also, C code is more likely portable to other microcontrollers with little or no modification [54].

Hence, it is evident that newly introduced data exchange protocols should avoid demanding additional features in regard to processing and memory. This provision is intended to prevent any impact on the required microcontroller. The thesis will consider both aspects, memory and processing power (see Part III) and the success of the developed exchange protocol will be judged according to a specific application. That said, the ISO/IEC 15118 communication between electrical and electronic equipment is meant to prove the practical relevance.

Chapter 3

Semi-structured Data in the Embedded Domain

So far the reader has come to know a bit about microcontrollers that are placed in various embedded domains taking part in the *Internet of Things* arena. Also, it sounds very promising that everyone's refrigerator at home places an order on the preferred online shop. Thus, the refrigerator may for example ensure that milk and butter is available for tomorrow's breakfast. However, let us change the point of view from the user perspective to the perspective of a provider of such a system. What are the actual technical consequences[1]?

To ensure a common understanding in regard to data exchange or respectively information exchange, the used terminology needs to be introduced first. The term *language* is subsequently used to define the notion and vocabulary of a given data exchange process while *format* stands for the actual representation of such a language. That said, different languages may have different capabilities and/or rules to express certain kind of information[2].

In the past up to these days, the exchange format has been well defined and mapped to the requirements whenever microcontrollers have been communicating with other microcontrollers or dedicated servers. Doing so guaranteed that on the one hand the microcontroller could be built small and cheap. On the other hand the server knew how to speak to any other embedded device. Typically the *language* was different depending on the actual device. That said, the controller of the refrigerator connecting to the Internet used a different vocabulary than the

[1] We will not elaborate on social consequences.

[2] One prominent example is HTML and XHTML that are main markup languages for creating Web pages. The syntax is very similar but there a few differences. In general XHTML syntax rules are far more rigorous than HTML. As such all elements and attribute names must appear in lower case.

sensor within the refrigerator tracking the currently available products. The task of the overall system is to combine all available information.

As one may imagine, building such a system is time-consuming and also very costly. The complexity rapidly grows the more devices start participating in the system. Replacing one device with another, that may imply a different representation of the actual data, becomes a nightmare. Furthermore, combining or merging one language with another might result in loosing information due to various language-specific restrictions.

This led to the situation that many organizations and groups have been starting to work on systems providing a *single* exchange language that can be used throughout the entire process. Moreover, existing and already successfully used concepts in nowadays web technologies such as the service-oriented architecture (SOA) have been mapped to the embedded domain [75]. To make clear that service-oriented architecture has been applied to microcontrollers and embedded devices it was often referred to as uSOA, μSOA, or ϵSOA [74].

SOA generally provides well-defined functionalities or services. SOA also defines how to integrate applications, and it can be seen as the continuation of older concepts of distributed computing. In practice, SOAs are build using web services standards (for example, SOAP[3] [55]) that have gained broad industry acceptance. However, SOA can be implemented using any service-based technology. The metadata to describe the characteristics of the services and also the data that is exchanged makes use of XML. XML is currently *the* semi-structured markup language.
Serge Abiteboul defined *semi-structured data* as follows: data that is neither strictly typed (i.e., not table-oriented as in a relational model) nor is the structure as rigid and regular as in standard database systems. The structure exists, is sometimes implicit and has to be extracted from the data [6].

This technical excursion to the world of nowadays microcontrollers and the current shortcomings state very well why there is the desire of a single exchange language. Chapter 5 will introduce XML as the chosen semi-structured data exchange language and will also highlight the drawbacks of XML when being used in the embedded domain. The generic term *binary XML* is generally used when speaking about various compact representations of XML that reduce the verbosity of XML documents and thereby also reduce the cost of parsing. However, in most of the cases binary XML solutions are applicable to certain restricted domains only. Due to restrictions of the afore mentioned formats, such as pre-shared knowledge and XML schema accuracy (see Chapter 6), communications islands have been built, frequently also with different binary XML formats in the overall system [40].

[3]SOAP (Simple Object Access Protocol) is a protocol specification for exchanging structured information for the implementation of Web Services in computer networks. It relies on the Extensible Markup Language (XML).

Chapter 4

Goal and Structure

The goal of this doctoral thesis is to elaborate a *single* binary XML format that is designed to be compatible with XML (at the XML Information Set level) and to expand its use to the embedded domain. The seamless integration of such a format into the existing XML stack is of huge importance. Additional improvements of the format are first highlighted, then thoroughly analyzed, and last the field-proven solution is presented (see Part III).

Smart grid is exemplarily taken as one example and is a form of electricity network utilising digital technology (see Figure 4.1). The term *smart grid* usually implies suppliers, energy storages and consumers exchanging energy (blue lines) on the one hand and communicating (red lines) on the other hand.

Figure 4.1: Smart Grid and Smart Energy[1]

[1]Picture copyright to E-Energy, http://www.e-energy.de/

So far, the power generation has been centralized. Large power plants have been producing power and made it available to consumers. The last couple of years, restructuring has been taking place. More and more decentralized parties have been involved in the overall system. Home photovoltaic systems have been widely used as well as small bio gas plants. The change between one-way interaction to two-way communications between suppliers and consumers demands an efficient way to exchange information in a standardized and flexible way. The information itself in this connection is also highly heterogenous ranging from sensor data to the extent of billing information. That said, the transparency between the actual data and the associated costs for a costumer is also very crucial.

A funding programme of the Federal Ministry of Economics and Technology (BMWi) in an interministerial partnership with the Federal Ministry for the Environment, Nature Conservation, and Nuclear Safety (BMU) has been launched to support relevant ambitions such as optimizing the energy supply system, encompassing everything from generation, and distribution right up to consumption. Moreover, the project is chartered to ensure more effective utilization of the existing supply infrastructure, expand the use of renewable energy resources, and reduce CO2 emissions[2].

The project has identified many different parties involved in such a smart grid communication that all demand to be equally supported. The variance extends from different devices as well as very different uses cases and kind of information. The extensive number of identified stakeholders asks for a standardized *and* single exchange protocol and format fulfilling various assembled demands.

This thesis aims to analyze the heterogeneous data and devices that are involved in such a scenario and tries to find the least common denominator to achieve efficient data exchange. Further, the research activities are influenced by technical problems that are to be solved before successfully applying the afore mentioned single exchange format to the relevant domains, such as the embedded domain.

One important aspect with regard to the embedded domain is the size of the exchanged data which needs to be kept at the very minimum. Metrics are introduced that describe the information that are to be exchanged primarily in terms of ratio between structure and actual data. Based on these defined metrics it is possible to predict the compression performance (percentage between original representation and compressed representation). Moreover, the metrics also highlight potential for further improvement. The research originated a theoretical method to describe the content or in other words the actual data. The method is used to realize an optimized datatype representation in a well-defined and also automatable way.

Other aspects of the work relate to the grammar system that builds the ba-

[2]E-Energy - Smart Energy made in Germany, `http://www.e-energy.de/en/`

sis for the co-developed and established exchange format. The grammars are analyzed and categorized according to their complexity. Proposals are given how to apply the different grammars to various systems ranging from powerful desktop/server machines down to highly restricted microcontroller-based architectures.

Yet another aspect of the thesis considers memory requirements and memory constraints while processing semi-structured data. Many query processors today work on an in-memory representation of the data. The in-memory representation can become very large and may render traditional processing infeasible on embedded and other resource-limited devices. Hence, a flexible concept is elaborated and also applied to the specific use case of processing data, which as a whole cannot be processed. The proposed mechanism allows sub-dividing the overall message in smaller chunks so that it allows memory-sensitive processing such as querying.

Besides the practical relevance of the thesis the research objectives and questions can be summarized as follows.

- Is it possible to determine a metric that categorizes semi-structured *binary* data?
- How can such a metric be used to predict data compression and processing time?
- How can the exchange format principles be algorithmically described?
- What is necessary to automatically map these principles (i.e., grammars) to running code or respectively to a processor of this format?
- What novel characteristics of the format may be advantageous for the embedded domain (e.g., runtime memory consumption)?

The approaches targeting the concrete compiled technologies and techniques are to be defined in a well-defined manner so that the results can be used, re-used and referenced in on-going scientific works or specifications.
Moreover, the research part of this work should establish a basis for researchers to look into the research topics more closely and to relate new research topics accordingly. The technology that has been analyzed is rather new and offers future potential for developing new aspects. Hence, the result that will be compiled are to be extendable, customizable, and algorithmically described to suffice future work.

The doctoral thesis is structured as follows.
The initial chapters of Part I give an overview about the involved technical areas. This overview constitutes the framework of the thesis and also stresses its

requirements and outcome. Furthermore, this leads to the definition of the term *embedded domain* that simply speaking comprises small controllers. The set of problems in the embedded domain regarding processing power and data exchange is given and alludes the need for an overall efficient exchange format with minimal requirements.

Part II introduces existing technologies meant to solve data exchange problems in nowadays use-cases by taking into account also the previously introduced restricted domain properties and the use of small microcontrollers. Measurements and analyses stress existing problems of nowadays used technologies and put the focus on a rather new technology that is subsequently introduced. As an active working group member of the World Wide Web Consortium (W3C) consortium and implementor of the de-facto reference implementation, I extensively helped to develop this new format named *Efficient XML Interchange* (EXI).

Based on this new EXI technology, optimizations and developments are proposed in Part III to make the format well applicable for the embedded domain. This implies theoretical aspects but also actual implementations.

Validating the applicability of the proposed solutions in the embedded domain and measuring their effectiveness conclude the work (see Part IV).

Part II

Overview of Semi-structured Data in the Embedded Domain

Chapter 5

XML Technologies

In recent years the need for supporting semi-structured data exchange in heterogeneous application areas has been raised due to the tremendous increase in communication devices. The solution to this problem has been XML [11] in most of the cases. The Extensible Markup Language (XML)[1] has been available for many years since it was first published as a W3C Recommendation in February 1998.

The success story of XML can be briefly summarized as follows. The format has been standardized by a well known and accepted consortium, the World Wide Web Consortium (W3C[2]). It offers a flexible and language-independent format in regard to application, platform, and programming language. It is meant to easily integrate in heterogeneous environments and has been very successful in doing so. XML is text-based, meaning that it is processable by machines and also readable for humans. The self descriptive and verbose format — the unmistakable tags — make it very easy to work with the data without actually knowing the exact structure.

Around the XML language itself, a very impressive XML stack with supporting technologies and standardized application programming interfaces (APIs) has been developed. Wide spread high quality libraries are available for XML schema languages, XML transformations, and XML query languages.

This chapter will introduce XML concepts and techniques that are necessary to understand XML applicability in the embedded domain. Moreover, XML metrics and actual real world test data is introduced that makes it possible to evaluate the applicability of different XML technologies.

[1] XML is extended from the Standard Generalized Markup Language (ISO 8879:1986 SGML)
[2] `http://www.w3.org/`

5.1 XML Fundamentals

5.1.1 XML and XML Schema

So far the XML markup language has been discussed without actually seeing a good example. Readers may or may not have seen XML documents in practice or used the XML language already. In either case, an actual example is very useful when starting to discuss a given technology. Listing 5.1 introduces an intuitive XML document that will be used throughout the work.

Listing 5.1: XML Notebook example

```
<?xml version="1.0" encoding="UTF-8"?>
<notebook date="2007-09-12">
    <note date="2007-07-23" category="EXI">
        <subject>EXI</subject>
        <body>Do not forget it!</body>
    </note>
    <note date="2007-09-12">
        <subject>shopping list</subject>
        <body>milk, honey</body>
    </note>
</notebook>
```

The simple XML document represents a `notebook` which in theory may contain an arbitrary number of `notes`. The given example contains two notes, one note about `EXI`, a technology we will read about later and another for a `shopping list`. This XML snippet shows very impressively why XML is so successful. Even without having further information about the kind of document, it is already possible to interpret the data and also to identify which information belongs to a given context.

Another important topic is XML data processing and its usage. XML (error-) handling or processing differs between *well-formedness* and *validity*.

XML Well-formedness

The XML specification defines an XML document as a text being *well-formed*, if it satisfies a list of syntax rules provided in the specification. The list is fairly lengthy [10]; some key points are:

- It contains only properly encoded legal Unicode characters.
- None of the special syntax characters such as "<" and "&" appear except when performing their markup-delineation roles.
- The begin, end, and empty-element tags that delimit the elements are correctly nested, with none missing and none overlapping.

- The element tags are case-sensitive; the beginning and end tags must match exactly. Tag names cannot contain any of the following characters:

  `!"#$%&'()*+,/;<=>?@[\]^`{|}~`

  nor a space character, and cannot start with -, ., or a numeric digit.

- There is a single "root" element that contains all the other elements.

XML Validity

In addition to being *well-formed*, an XML document can be *valid*. The validity of a document can be checked by means of a schema language. There is more than one schema language, also specified by different organizations and working groups. As part of the XML specification there is the possibility to use Document Type Definitions (DTD)[3] that declare precisely which elements and references may appear where in the document of the particular type, and what the elements' contents and attributes are.

XML schema, published as a W3C recommendation in May 2001, is another XML schema language that was designed to offer features that were not available in XML's native DTD such as namespace awareness and datatypes (ability to define element and attribute content as containing values such as floats). Due to these additional features, the fact that it was designed by the W3C, home of XML, and due to its pervasiveness in the XML community we subsequently focus on XML schema [76, 67] as *the* schema language.

XML processors are classified as validating or non-validating depending on whether or not they check XML documents for validity. Although XML well-formedness is sufficient for many use cases it may also cause issues in other environments. Let us use the notebook XML example and add additional information about who is the one the note is belonging to, in form of an attribute `belongsTo`. In respect to well-formedness this does not break XML processing. In respect to a given application that is only meant to display given information of a `note`, e.g., on your mobile phone, it may cause problems. In a best-case scenario, the application just ignores the additional information. More probably, the application refuses to display the information properly, given that it was neither meant nor built to show an attribute `belongsTo`.

Listing 5.2 introduces one possible XML Schema Document (XSD) which describes the root element, its possible children, and contents datatypes.

In this XML schema document we have one global element `notebook`. Global elements are elements that are immediate children of the `schema` element. Local elements are elements nested within other elements or complex types. Only global

[3] XML uses a subset of SGML DTD

Listing 5.2: XML schema example for notebook XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
    <xsd:element name="notebook">
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element name="note" type="Note"/>
            </xsd:sequence>
            <xsd:attribute ref="date"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="Note">
        <xsd:sequence>
            <xsd:element name="subject" type="xsd:string"/>
            <xsd:element name="body" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute ref="date" use="required"/>
        <xsd:attribute name="category" type="xsd:string"/>
    </xsd:complexType>
    <xsd:attribute name="date" type="xsd:date"/>
</xsd:schema>
```

elements can be the root element of a valid XML document[4] and can be referenced by other complex types. The element `notebook` may consist of an arbitrary number of `note` elements (specified by `maxOccurs="unbounded"`) while containing at least one `note` element (not present in the schema definition but implicit according to the default value `"1"` for the schema attribute `minOccurs`). We further have one global attribute `date` that is referenced from the anonymous complexType describing the element `notebook` and also from the the complexType `Note`. The attribute `date` is typed as XML schema datatype `xsd:date` while the remaining elements and attributes are of the type `xsd:string`. The complexType `Note` is a sequence of elements `subject` and `body`, requiring an attribute `date` and allowing an attribute `category`.

The XML schema definition differs between complex and simple type definitions. While complex type definitions may consist of sub-elements, a simple type definition may only consist of a simple datatype.

The XML schema language, also known as XSD (XML Schema Definition), provides a set of 19 primitive datatypes (anyURI, base64Binary, boolean, date, dateTime, decimal, double, duration, float, hexBinary, gDay, gMonth, gMonthDay, gYear, gYearMonth, NOTATION, QName, string, and time). It allows new data types to be constructed from these primitives by three mechanisms:

[4]For completeness it is worthwhile to note that there is one exception to this statement when it comes to a non-global element that uses xsi:type to achieve XML validity.

- restriction (reducing the set of permitted values)
- list (allowing a sequence of values)
- and union (allowing a choice of values from several types)

25 derived types are defined within the specification itself[5], and further derived types can be defined by users in their own schemas.

XML Drawbacks for Resource-limited Environments

XML drawbacks such as verbose data exchange and/or untyped textual data are undesirable if not unacceptable for resource-constrained environments. These properties make an exchange format such as XML hardly applicable for small device classes. Resource-limited environments generally dispose of limited transmission capabilities (bandwidth) and limited processing speed. Further, low costs in regard to battery, hardware, transmission, and energy are crucial for the success of such systems. These requirements led to develop new XML encoding formats.

5.1.2 XML Optimization Fields

Doubtlessly the success of XML has been enormous in areas where efficient data exchange or processing is not crucial. Especially in resource-restricted environments, XML implies properties that are infeasible due to its text-based nature.

When dealing with data exchange formats, two important aspects are data compression and processing time. Both aspects generally relate to typed data and the verbosity or non-verbosity of a given format. XML's flexibility and openness comes along with undisputable verbosity. In addition, the text-based format increases processing time. This not only applies to element or attribute tag names but also to the actual data. A float data value represented as the string `"12.5"` needs to be converted into an in-memory representation of a float, which is time consuming. Generally speaking one could argue that XML was not designed for performance-sensitive environments.

When it comes to efficient XML handling we differentiate between fast processing of XML and a compact representation.

Processing Time

The term *XML processing* denotes reading and writing XML information. In this context *deserialization* or sometimes also *decoding* is used for processing the actual data. The opposite operation, *serialization* or *encoding* translates data into a format that can be stored (e.g., in a file or a byte buffer).

[5] `http://www.w3.org/TR/xmlschema-2/#built-in-datatypes`

XML Screamer [42] is one technique that manages to optimize processing time in a system in which customized high-performance XML parsers are prepared using parser generation and compilation techniques. Parsing is integrated with schema-based validation and deserialization, and the resulting validating processors are usually significantly faster than traditional non-validating parsers. Moreover, high performance is achieved by integration across layers of software that are traditionally separate by avoiding unnecessary data copying and transformation. Unfortunately, XML Screamer does not provide a solution for compact representation.

Data Compression

Compact data representation is the other crucial aspect. General purpose compression techniques that do not necessarily target XML documents per se are providing a good solution for compressing data. The compression can be applied to any document such as images, videos, or text-based formats as in the case of XML. Nevertheless, in regard to processing, it makes the situation worse. The requirements of processing compressed data is higher than processing plain XML documents. Additionally to processing XML, the data needs to be packed on encoder side and un-packed on decoder side.

Depending on the actual use case various optimization efforts (e.g., XML Screamer and general purpose compression techniques) led to either good processing performance or a compact data representation. However, more challenging application fields originated new formats taking into account both relevant aspects. These formats share the common XML information set (expressivity in regard to XML features) but give up intrinsic properties of XML, namely the text-based nature and human-readability. These formats are usually summarized by the term *Binary XML*. Chapter 6 introduces some aspects and various flavors of binary XML formats.

5.2 XML Content Density

The XML community uses several metrics to describe properties and/or the complexity of XML documents. Some metrics take into account the XML depth, in other words the maximum level of nested XML elements to quantify structural properties of XML documents. Others determine the complexity of XML documents based on syntactic and structural aspects [68]. And yet others consider also available XML schema knowledge.

To facilitate comparisons between the work done in W3C working groups [81] and the measurements presented in this thesis, the same metric is used. The met-

ric in use is XML *content density* (the ratio between *text* and *markup*, abbreviated CD). The CD is based on XML characters and is computed as follows:

1. Gather all character data information items that are the direct children of an element information item.
2. Gather all the values of all the attribute information items.
3. Sum up the size in characters of the text data gathered in the previous two steps.
4. The content density is the ratio of the sum in the previous step and the size of the entire document in characters.

The following example explains the metric by means of an example XML document.

Example 5.2.1 (Content Density Computation for XML document in Listing 5.1)**.**

1. *Data information items are "EXI", "Do not forget it!", "shopping list", and "milk, honey": 44 characters.*
2. *Attribute information items are "2007-09-12", "2007-07-23", "EXI", and "2007-09-12"): 33 characters.*
3. *In the previous two steps 77 characters have been gathered.*
4. *The size of the entire document in characters is 320.*
$CD = \frac{77}{320} = 24\%$

5.3 XML Test Data Characteristics

This section describes characteristics of the given test data, which subsequently is used throughout the thesis to evaluate the performance of various XML technologies in regard to data compression and processing performance. The value of measurements and comparisons highly depend on the test data that is used. Hence, a good coverage of relevant test-cases and a valid and sensible methodology to quantify the data is crucial.

The previously mentioned W3C activities in the binary XML area produced a collection of more than 10000 XML documents of various application areas or so called test groups. Each group out of the 20 test groups (e.g., WSDL, SVG) consists of XML document instances matching the same vocabulary or related applications. Because a smaller set is easier to manage for further analyses, the W3C working group selected a representative sub-set of XML instances, taking

into account test data from all test groups and also considering test instances with diverse content density[6].

In summary, the documents in the test suite are characterized along two axes: size in bytes and content density. These characteristics are largely independent of each other, since content density is measured as a percentage. Figure 5.1 depicts the result values for the selected 88 XML test documents. The test documents range from 99 Bytes up to 70 MegaBytes. Moreover, the content density distributes from nearly 0%[7] up to almost 100%. These numbers confirm once again that we deal with a high variety of test documents with different XML flavor in regard to size and content density.

The results in Figure 5.1 match with the measurements presented in [81] besides some errors (the calculated content density was inaccurate for two test groups) that have been detected and reported back to the working group. The EXI working group acknowledged the issues by creating an errata page[8].

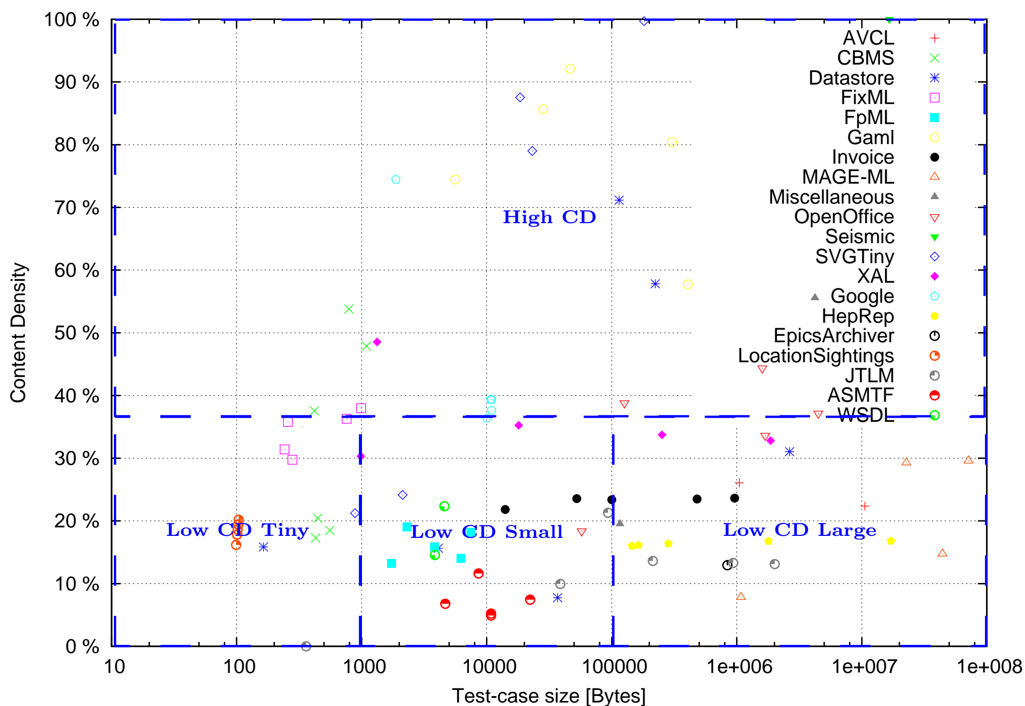


Figure 5.1: XML Content Density

Please note that given to the wide variety of test documents most detailed

[6]The test instances are available for download through the W3C measurement test framework, see http://www.w3.org/XML/EXI/#TestingFramework

[7]The content density value of 0% originates from a test document out of the JTLM test group which is composed of elements only without any character nor attribute information items

[8]Measurements note errata, http://www.w3.org/XML/EXI/wiki/MeasurementsNoteErrata

measurements throughout the thesis use four analysis groups or clusters to further sub-divide the 88 XML test documents. The basis for comparing is the size in bytes and the according XML content density. Figure 5.1 depicts these clusters with dashed boxes.

Cluster #1: High XML Content Density (20 documents)

This cluster consists of documents having a content density higher than 38%. Due to the high ratio of text (attribute information items and/or character data information items) compared to XML markup, these documents behave similarly.

Cluster #2: Low XML Content Density - Large documents (24 documents)

This cluster consists of documents having a content density less than or equal to 38% and a size of more than 100 kilobytes.

Cluster #3: Low XML Content Density - Small documents (23 documents)

This cluster consists of documents having a content density less than or equal to 38% and a size between 1 and 100 (inclusive) kilobytes.

Cluster #4: Low XML Content Density - Tiny documents (21 documents)

This cluster consists of documents having a content density less than or equal to 38% and a size less than or equal to 1 kilobyte.

Note that documents from the same test group do not always belong to the same cluster (e.g., the test group *Datastore* has test cases in all four clusters). Table 5.1 reports how many test documents of a given test group falls into each cluster.

An extensive description of each test group can be found in the measurements note [81] published by the EXI working group.

Test Group	High CD	Low CD Large	Low CD Small	Low CD Tiny
AVCL		2		
CBMS	2			4
Datastore	2	1	2	1
FixML				5
FpML			5	
GAML	5			
Invoice		3	2	
MAGE-ML		4		
Misc	1	1		
OpenOffice	2	2	1	
Seismic	1			
SVGTiny	3		1	1
XAL	1	2	1	1
Google	3		2	
HepRep		5		
EPICS		1		
LocationSigthings				8
JTLM		3	2	1
ASMTF			5	
WSDL			2	

Table 5.1: Test-groups classified by number of XML documents per cluster

Chapter 6

Binary XML

So called *Binary XML* formats refer to any specification which define a compact representation of XML and thereby reduce the verbosity of XML documents and the cost of parsing. There are several competing formats, none in the past has been widely adopted by a standards organization or accepted as the de facto standard.

6.1 Related Work

The literature usually distinguishes between three possible approaches: general purpose, XML-aware, and schema-aware techniques [40].

General Purpose Compression Techniques

General purpose compression techniques do not necessarily target XML documents. The compression can be applied to any document such as images, videos or text-based formats as in the case of XML.

While software applications such as WinZip, WinRar, and 7-Zip are well known, the actual compression file formats (e.g., zip, tar, bzip2) are less prominent. The term *gzip* usually refers to the GNU Project's implementation, "gzip" standing for GNU zip. It is based on the DEFLATE [23] algorithm, which is a combination of Lempel-Ziv (LZ77 [82]) and Huffman [38] coding.

The major advantage of such compression techniques, in regard to XML, is its high availability and its wide adoption. On most of the platforms general purpose compressors are available out of the box. Also, they are usually free software.

The problem of good compaction and the resulting smaller documents is that this positive characteristic comes along with additional processing, meaning that general purpose compression techniques not only compress data but also introduce

an additional processing step (see Figure 6.1). In the case of a general purpose or sometimes also called traditional compression technique an in-memory XML representation is first transformed to an XML document before it can be compressed. XML-aware compressors are capable to directly *stream* XML through a dedicated XML application programming interface (API).

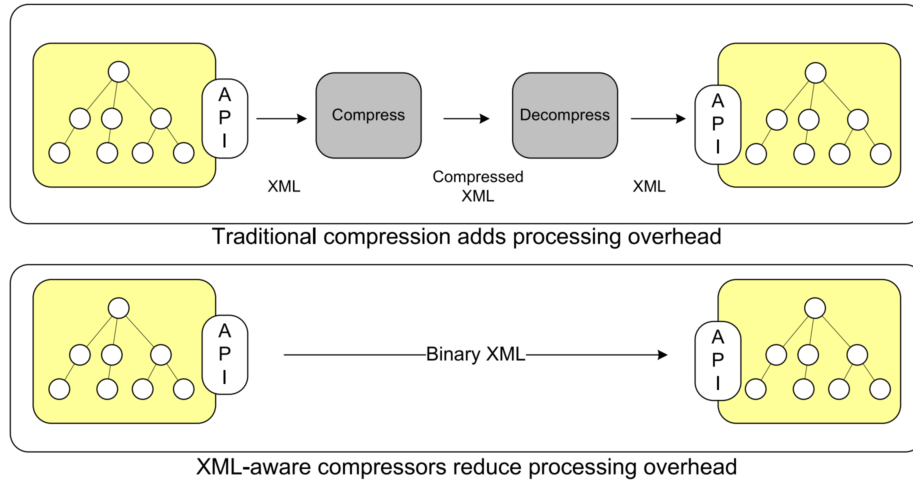


Figure 6.1: Processing steps when compressing XML

XML-aware Compressors

XML-aware compressors take into account the nature of XML such as the knowledge about XML elements, attributes, namespaces, and other XML constructs such as processing instructions (e.g., <?xml version="1.0"?>) or comments (e.g., <!- This is a comment ->).

XMill and WBXML are two XML-aware compressors that will be discussed shortly. This is because both technologies introduced concepts that had and still have significant impact on other XML compressors.

Schema-aware Techniques

An even more satisfying approach is the use of XML schema knowledge. An XML schema document describes the structure of an XML instance and also identifies the types of given elements and attributes. This leads to the fact that the markup of an XML instance can be represented very efficiently. The tag names are known and can be pre-indexed and shared between sender and receiver and also the data itself can be represented in a typed fashion (e.g., xsd:integer as integer).

This approach usually combines fast processing with high compaction of the data. Many known schema-aware formats can only work with proper XML schema

knowledge. This means that without the *corresponding* schema information an XML instance cannot be encoded.

In many cases it is very difficult to clearly characterise a binary XML format according to the three possible XML compression approaches. For example, depending on the used format features and options, a binary XML format may belong to XML-aware compressors or to a schema-aware technique.

Subsequently a selection of XML compressors is given. The selection is based on the acceptance in important environments such as the telecommunication and on technical concepts that are of interest in future sections of this work.

6.1.1 XMill

Hartmut Liefke and Dan Suciu published in the year 1999 a very interesting approach about an efficient XML compressor called XMill. It is claimed that the tool achieves about twice the compression ratio of gzip at roughly the same speed [49].

The compressor, called XMill, incorporates and combines existing compressors. The basic idea is to make use of technologies such as gzip with the knowledge that such compression techniques work best for similar data.

The compressor applies three principles. First it separates XML structure from XML data. The structure, consisting of XML tags and attributes, is compressed separately from the actual XML data. The compressed structure stream essentially consists of a sequence of string items representing element and attribute tag names. Second the data items, element content and attribute values, are grouped into containers. Each container is compressed separately. With reference to the previously introduced XML document in Listing 5.1, all XML character information items belonging to the element `subject` (i.e., "EXI" and "Shopping List") form one container, while all attribute information items belonging to the attribute `date` (i.e., "2007-09-12", "2007-07-23" and "2007-09-12") form a second container. Third specialized compressors, so called *semantic compressors*, are used for various containers. Hartmut Liefke and Dan Suciu state in their work [49] that setting a user-definable compressor (e.g., gzip, bzip2) may increase compression performance given that for example some data items are text while other items are numbers.

The approach is working well to better utilize network bandwidth or for data archiving, where the goal is to reduce space requirements. It is less useful when also reducing processing time is important or data streaming is required. Moreover it has been shown that XMill wins over existing compression technologies only if the data set is large, typically over 20 KB [49]. Hence it is of limited use where many small-sized XML messages are to be exchanged or processing time is crucial.

6.1.2 WBXML

WAP Binary XML (WBXML) [51] is a binary representation of XML to allow XML documents to be transmitted in a compact manner over mobile networks. It was developed by the WAP (Wireless Application Protocol) Forum, which later was aggregated with other industry forums to the Open Mobile Alliance (OMA) as a standards body, which develops open standards for the mobile phone industry. Moreover it was submitted to the World Wide Web Consortium[1].

WBXML is used by a number of mobile phones where it is for instance used for transmitting address book and calendar data (SyncML).

The WBXML specification is is often referred to as a *tokenization* format with a built-in state machine meaning that assigned identifiers are used for some symbols or tokens. The strings that are tokenized in WBXML comprise namespace URIs and prefixes, element and attribute names and also character and attribute values. Tokens can be assigned either dynamically or statically. Dynamic assignments discover the tokens as they appear in the serialized documents and are encoded much more efficiently after the first appearance. Static assignments require that the tokens need to be known beforehand to communicating parties. Moreover it is also possible to establish a session where the token assignment persists from one message to the next message.

6.1.3 ASN.1

ASN.1 [59] is a ISO/IEC and ITU-T standard and constitutes a flexible notation that describes data structures for representing, encoding, transmitting, and decoding data [62, 63]. It is a proven specification and widely used in the telecommunication and computer networking sector.

ASN.1 together with specific ASN.1 encoding rules facilitates the exchange of structured data. The standard ASN.1 encoding rules include, among others, XML Encoding Rules (XER), Basic Encoding Rules (BER) [60] and Packed Encoding Rules (PER) [61]. PER encoding rules are meant to produce a compact transfer syntax for data structures described in ASN.1. The XML Encoding Rules (XER), in turn, attempt to bridge the gap between textual encoding of data structures defined using ASN.1 notation and ASN.1 itself.

6.1.4 FastInfoset

The FastInfoset specification [69] is defined by both the telecommunication division of the ITU and the ISO standards bodies as an alternative to the XML document format. FastInfoset aims to optimize both document size and processing performance and mostly evolved out of the initiative at Sun Microsystems

[1] http://www.w3.org/TR/wbxml/

identifying performance problems in existing implementations of Web Services standards [70].

Similar to WBXML, FastInfoset is a tokenization format that supports dynamic tokenization but also permits static token assignment. In contrast to many tokenization formats that are limited to a certain number of tokens, FastInfoset uses an in theory indefinitely-increasing counter for their token values. Hence, FastInfoset uses an encoding for integers that permits any integer to be represented. This representation is often referred to as variable-length integer encoding[2] that allows storing an integer in a variable number of bytes. Small integers require smaller number of bits and every token persists until the end of the coding process.

Interesting to note is that FastInfoset, often abbreviated to FI, is in fact an application of ASN.1 and is formally specified using ASN.1 formalisms and Encoding Control Notation (ECN) encoding rules [69].

6.1.5 BiM

BiM (Binary MPEG format for XML) is another international standard defining a generic binary format for encoding XML documents. The binary MPEG format for XML relies on schema knowledge between encoder and decoder in order to reach high compression efficiency while also providing flexibility in regard to XML fragmentation. The XML standard supports logical documents composed of possibly several entities but it may be desirable to process one or more of the entities or parts of entities while having no interest, need, or ability to process the entire document [80]. Such parts are referred to as XML fragments.

Moreover, BiM defines means to compile and transmit schema knowledge information to enable the decoding of compressed XML documents without a priori schema knowledge at the receiving side [52, 36, 57, 77].

BiM is used for example as the standard binary format for XML encoding in the following technical specifications:

- MPEG-4 Part 20 or MPEG-4 Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)
  ISO/IEC 14496-20 [5, 24]

- MPEG-7 Systems
  ISO/IEC 15938-1 [52]

- MPEG-21 Binary Format
  ISO/IEC 21000-16 [39]

[2] Usually we differentiate between two variable-length integer encoding techniques. One possibility is to store the length of the value as prefix and the the other possibility is to use a *continuation* bit in each byte that indicates whether another byte is following.

- TV-Anytime
  ETSI TS 102 822 [25]

- Digital Video Broadcasting (DVB)
  ETSI TS 102 323 [28], ETSI TS 102 539 [26], ETSI TS 102 471 [27]

6.2 Efficient XML Interchange (EXI) Format

Many binary XML technologies were developed and specified in the past to overcome the problems and use cases that have been identified with regard to XML in restricted environments.

Some of the evolving formats are general purpose compression techniques (e.g., gzip), some XML-aware compressors (e.g, XMill, WBXML), others schema-aware techniques (e.g., ASN.1, BiM), and yet others combinations of the mentioned solutions (e.g., FastInfoset). Nevertheless, none of the techniques has been used or selected as *the* efficient exchange format for XML by the community. Each has its right to exist in a rather limited application range only.

In the year 2004, the Word Wide Web consortium, home of XML, has tasked the XML Binary Characterization (XBC) working group to collect the demands for a *single* binary XML format. The outcome was a list of properties [18], use cases [19], measurements methodologies [35], and a XML binary characterization [33] guiding to a single format solving the problem where the overhead of generating, parsing, transmitting, storing, or accessing XML-based data may be deemed too great for a particular application. The conclusions of the XML Binary Characterization working group can be summarized as follows [33]:

- Binary XML is needed.

- Binary XML is feasible.

- The W3C must produce Binary XML.

- Binary XML must integrate with XML[3].

Based on these conclusions, 18 extensive use cases, and 38 different format properties and considerations, the EXI working group was chartered in 2005, the same year XBC was closed. The Efficient XML Interchange (EXI) working group is part of the W3C XML Activity and followed the XBC work. The main objective of the EXI working group was to develop a format that allows efficient interchange of the XML Information Set. The goals of the working group were:

[3]Binary XML must integrate with the existing XML stack and not require changes to XML itself [33].

1. Fulfill the design goals of XML[4] with the following exceptions:
   a) The interchange format must be compatible with the XML Information Set [78, 79] instead of being "compatible with SGML" (XML goal 3).
   b) For performance reasons, the format is not required to be "human-legible and reasonably clear" (XML goal 6).
   c) Terseness[5] in efficient interchange is important (XML goal 10).
2. Address all requirements and use cases from the XML Binary Characterization Working Group.
3. Maintain the existing interoperability between XML applications, as well as XML specifications.
4. Establish sufficient confidence in the proposed format, in particular establish confidence that the performance gains are significant, and the potential for disruption to existing processors is small.

The EXI working group started by considering existing solutions and evaluated each in terms of implementability and performance. As a result of the measurements [81], the working group selected Efficient XML[6] to be the basis for the proposed encoding specification to be prepared as a candidate W3C Recommendation. Follow-up work has centered around integrating features from the other considered solutions, particularly variations for both more efficient structural and value encodings [81].

In the following the main EXI format specification [73] concepts and ideas are discussed.

6.2.1 Basic Concepts

The EXI specification [73] describes the basic concepts as follows.

> *"EXI achieves broad generality, flexibility, and performance, by unifying concepts from formal language theory and information theory into a single, relatively simple algorithm. The algorithm uses a grammar to determine what is likely to occur at any given point in an XML document and encodes the most likely alternatives in fewer bits. The fully generalized algorithm works for any language that can be described by a grammar (e.g., XML, Java, HTTP, etc.); however, EXI is optimized specifically for XML languages."*

[4] `http://www.w3.org/TR/2004/REC-xml-20040204/#sec-origin-goals`

[5] When creating XML element names, `first_name` is better than `fname` because it's clearer and more human readable. Keeping element names short should not sacrifice human-readability.

[6] `http://www.agiledelta.com/w3c_binary_xml_proposal.html`

EXI Event Type	Grammar Notation	Event Content
Start Document	SD	
End Document	ED	
Start Element	SE (qname) SE (*) SE (uri : *)	qname
End Element	EE	
Attribute	AT (qname) AT (*) AT (uri : *)	qname, value
Characters	CH	value
Namespace Declaration	NS	uri , prefix , local-element-ns
Comment	CM	text
Processing Instruction	PI	name, text
DOCTYPE	DT	name, public, system, text
Entity Reference	ER	name
Self Contained	SC	

Table 6.1: EXI events

In this context, an EXI stream is an EXI header followed by an EXI body. The EXI body carries the content of an XML instance, while the EXI header communicates the options used for encoding the EXI body.

EXI disposes of a set of EXI events that may occur in an EXI stream. Table 6.1 depicts all event types, their associated grammar notation, and the associated event content, if any. At this point, the representation of the event content item *value* is of interest. A *value* item represents attribute or character data and uses a string representation if no schema information is available, or is represented by its associated (schema) datatype if available[7].

The concept of a qualified name is of interest for further understanding. A qualified name, or sometimes also qname/QName, defines a valid identifier for elements and attributes. QNames are formally described by the W3C[8] as:

```
QName ::= PrefixedName | UnprefixedName

PrefixedName ::= Prefix ':'  LocalPart

UnprefixedName ::= LocalPart
```

The Prefix is used as placeholder for the namespace URI and the LocalPart as the local part of the qualified name. A local part can be an attribute name or an element name.

[7] EXI settings may overwrite the default representation
[8] `http://www.w3.org/TR/REC-xml-names/#NT-QName`

The sequence of EXI events in Figure 6.2 can be easily mapped to the structure of the XML document shown in Listing 5.1. Every document begins with a Start Document (SD) and ends with an End Document (ED). We additionally have Start Element (SE) and the associated End Element (EE) events, Characters (CH) and Attribute (AT) events with the associated event content. Grey buckets represent structure information and colored buckets are used for content information. The color is determined by the associated qname (e.g., date, category, subject, body). [66]

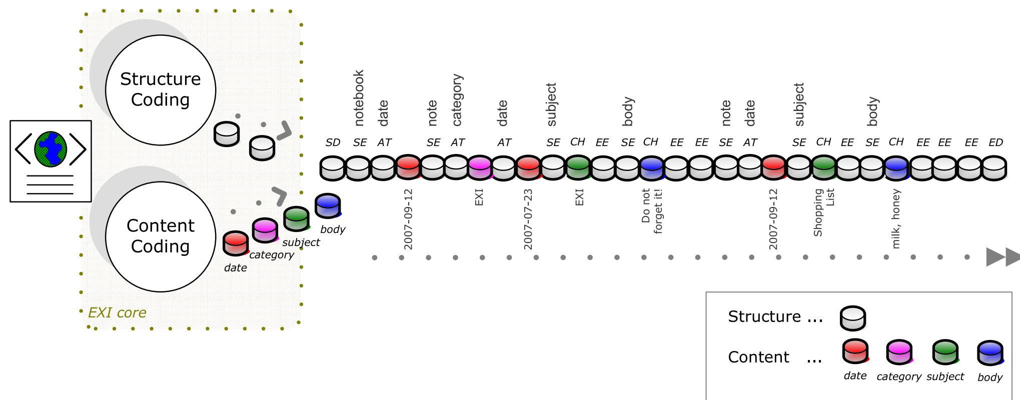


Figure 6.2: EXI Body stream for Listing 5.1

6.2.2 EXI Grammars

EXI is a grammar-based format, meaning that its knowledge-based encoding uses a set of grammars. Grammars describe what is most likely to occur at any given point in an EXI stream. Some grammars are very generic and applicable to any XML document, fragment, and element. These are called *built-in XML grammars*. Other grammars are derived specifically from XML schema knowledge and are therefore called *schema-informed grammars*.
The two kinds of grammars may be used in combination with each other and essentially behave the same. The only difference is that built-in grammars may evolve over time to adapt themselves to a given XML instance while schema-informed grammars remain the same over time. Hence, schema-informed grammars can be re-used for various XML instances.

In EXI terminology, an EXI grammar consists of a set of grammar productions. A grammar production, in turn, consists of an in this context unique event (e.g., StartElement event) and may lead to another grammar. Given that XML is not a regular language [66], a single grammar cannot be used to represent an entire XML event stream. Instead, an EXI coder uses a stack of grammars, one for each element content model (just like an XML schema validator might do).

Built-in XML Grammars

Figure 6.3 depicts a built-in element grammar. We differ between StartTag and Element content grammars. The StartTag grammar deals with events that happen right after an element starts, such as namespace declarations and/or attributes. Once, for instance, character events or another start element appears, the current state moves on to an Element content grammar. An Element grammar anticipates less events and hence less options that lead to less bits to indicate an event transition.
The figure also outlines the stack of grammars showing a layer for DocContent, SE(notebook), and SE(note) grammars.

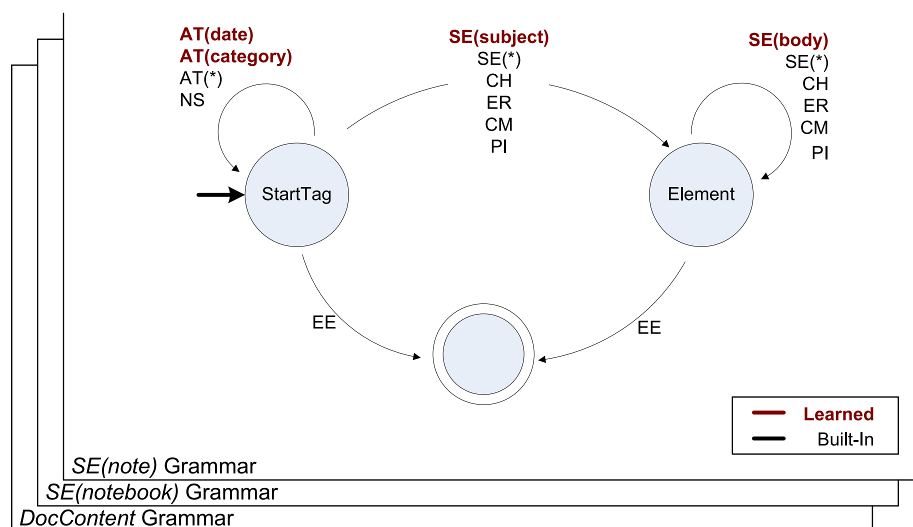


Figure 6.3: Built-in element grammar for SE(note)

Moreover, the figure also outlines the evolution of built-in grammars. At the instantiation phase of a built-in element grammar a small set of very generic grammar events are available (e.g., AT(*) standing for any attribute and SE(*) for any start element). When processing an XML instance more knowledge from a given XML instance is retrieved and also used for further processing. An actual XML instance adds dedicated events for attributes, start elements, as well as for end element events. For instance Listing 5.1 leads to the following learned events for the element `note`:

- AT(date)
- AT(category)
- SE(subject)
- SE(body)

The reason for doing so is to better reflect the actual element for any subsequent appearance. A second or third element `note` re-uses the learned events and hence does not require to code the actual local name or namespace URI over and over again.

An EXI processor disposes of a set of global grammars that are matched and re-used according to the associated qualified name (e.g., `note`).

Schema-informed Grammars

Schema-informed grammars, in contrast to built-in grammars, do not evolve over time. Figure 6.4 illustrates a schema-informed grammar for the element `note` which in turn is of complexType `Note` (see Listing 5.2). EXI requires to sort attributes in lexicographical order[9] which on the one hand results in less grammar states and on the other hand improves data compression. Moreover, it is assumed that XML schema expresses the rules to which an XML instance conforms. Hence there is no need for grammar evolution which makes it also possible to build schema-informed grammars once for multiple encoding and decoding processes. XML information that do respect XML schema information can still be represented by schema-informed grammars but less efficient (for reasons of simplicity the figure does not illustrate these grammar states and transitions).

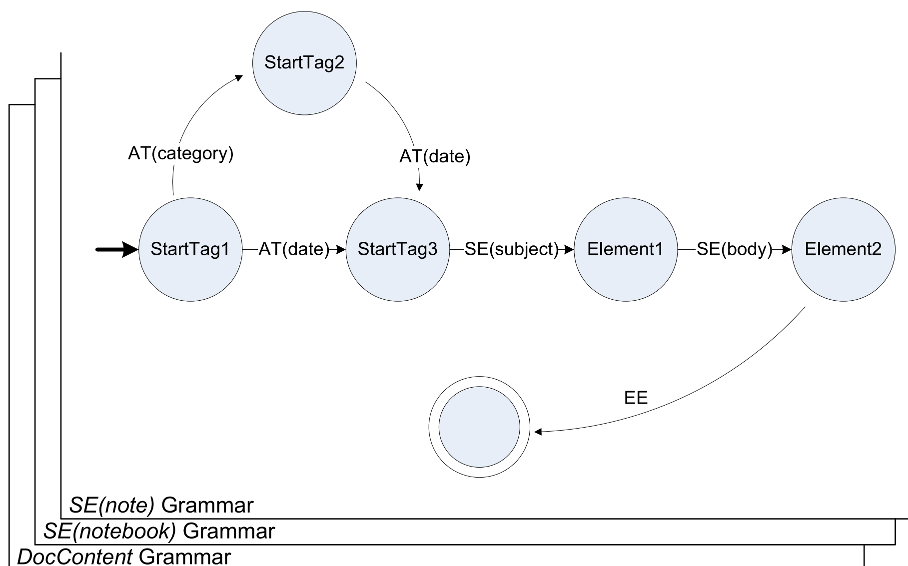


Figure 6.4: Strict schema-informed grammar for StartElement(note)

[9]When in EXI it is stated that strings are sorted in lexicographical order, it is done so character by character, and the order among characters is determined by comparing their Unicode code points. In the case of *qnames* sorted lexicographically means first by local-name, then by uri. [73]

The thesis does not go into too much details about how schema information are to be used to create schema-informed EXI grammars. The EXI specification [73] explains the process in a very detailed manner. Nonetheless, we will come back to grammars and how grammars are processed and used in an efficient manner later in this thesis (see Chapters 9 and 10).

Grammar Event Codes

EXI processors represent a given event such as a start element or an attribute by indicating the appropriate event serializing an event code first followed by the according event content. Each event code is represented by a sequence of one, two, or three parts that uniquely identifies an event (see event code 2.7.0 for CM production in Figure 6.5).

Each grammar production in an EXI grammar is linked with a unique *event code* in this given context that approximates the likelihood the associated production rule will be matched in regard to other productions.

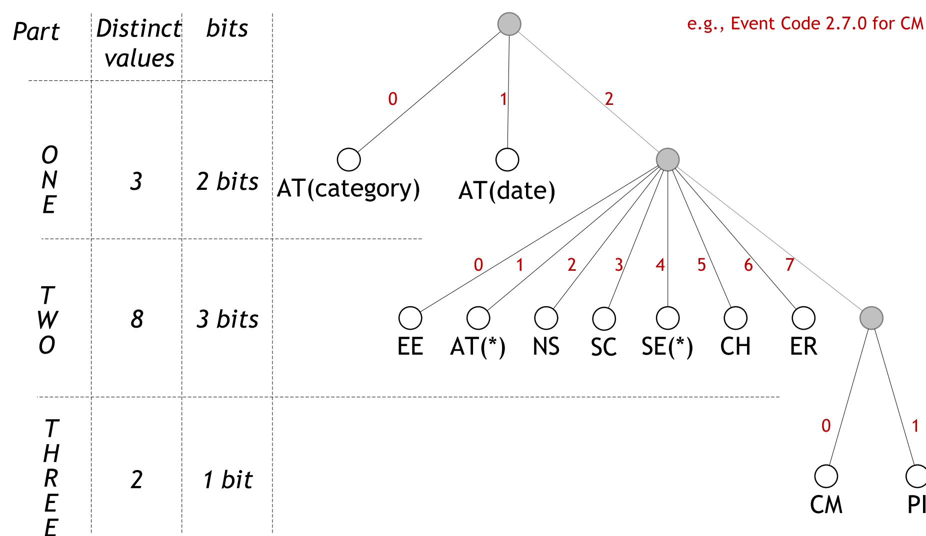


Figure 6.5: Grammar Event Code Tree for grammar state *StartTag1* in Figure 6.4

An EXI event code tree is similar to a Huffman tree [38] in that shorter paths are generally used for symbols that are considered more likely. However, event code trees are much simpler. While a Huffman tree in regard to its depth is usually unrestricted, an event code tree is shallow and contains at most three parts, where each part is a non-negative integer.

Event codes in an EXI grammar are assigned to productions in such a way that shorter event codes are used to represent productions that are more likely to occur (i.e., AT(category) and AT(date)). Conversely, longer event codes are used to represent productions that are less likely to occur. EXI grammars are designed

in a way that the average number of bits needed to represent each production or respectively the according event code is less than that for a grammar in which more likely and less likely productions are not distinguished. The following Table 6.2 illustrates this principle via a comparison between a naive event code assignment example and the approach EXI uses for event code assignment.

Table 6.2: *Naive* Event Code Assignment vs. EXI Event Code Assignment

Event	Indicator	#bits	Part	Event Code			#bits
AT(date)	0	4	One	0			2
AT(category)	1			1			
EE	2		Two	2	0		2 + 3
AT(*)	3			2	1		
NS	4			2	2		
SC	5			2	3		
SE(*)	6			2	4		
CH	7			2	5		
ER	8			2	6		
CM	9		Three	2	7	0	2 + 3 + 1
PI	10			2	7	1	
#distinct values	11			3	6	2	

On the left side of the table, where productions are not separated according to their probability, a 4-bit indicator code is needed to represent each entry. On the right hand side of the table, on the other hand, code lengths vary from 2 bits to 6 bits since productions are grouped based on their likelihood to occur. Assuming the content model for the element being encoded corresponds to the sequence AT(category) and AT(date) (i.e., the element declares two attributes) then the encoding of all the event codes will be 4 bits shorter using the second table.

Figure 6.5 and Table 6.2 depict the initial schema-informed grammar state "StartTag1" grammar event code tree with the full set of productions for the element `note`. The EXI specification enables or disables the capacity for the preservation of a certain type of information. One can configure the so called *EXI fidelity options* (see Table 6.3) that may prune events that are not required from the grammars, improving data compression and processing efficiency. Hence, applications can use the preserve option to specify the set of fidelity options they require. This may even lead to prune all brunches from part two and three (e.g., EE, AT(*), NS, SC, SE(*), CH, ER, CM, and PI) so that grammar productions shrink to one part or respectively to one event code part only. This is the case for the EXI option *strict*, using a strict interpretation of the schemas and omitting preservation of fidelity options, such as comments, processing instructions, namespace prefixes and any deviant information from the schemas.

Table 6.3: Fidelity options

Fidelity option	Default Value	Effect
Preserve.comments	false	Productions of CM events are pruned from grammars
Preserve.pis	false	Productions of PI events are pruned from grammars
Preserve.dtd	false	Productions of DOCTYPE and ER events are pruned from grammars
Preserve.prefixes	false	NS events are pruned from grammars and namespace prefixes are not preserved
Preserve.lexicalValues	false	Lexical form of element and attribute values is not preserved

6.2.3 EXI String Table

EXI uses a string table to assign "compact identifiers" to string values. String values found in the string table are represented using the associated compact identifier instead of representing the string "literally" again. Some content items (see Table 6.1) are encoded using a string table:

- uris
- prefixes
- uri and local-name in qnames
- values

The string table is initially pre-populated with string values that are likely to occur and is dynamically expanded to include additional strings while processing an actual document.

The EXI string table is organized into partitions (see Figures 6.6 and 6.7), namely *uri*, *prefix*, *local-name*, and *value* partition. One can see that an additional string entry in the *uri* partition does not affect the *value* partition and vice-versa. This allows the identification of each entry in a given partition with a short code. Moreover, depending on the purpose of the partition, each partition is optimized for frequent use of either compact identifiers or string literals. Uri and prefix content items are expected to contain a relatively small number of entries, used repeatedly throughout the document, and are optimized for the frequent use of compact identifiers. Local names and all string value content items are optimized for the frequent use of string literals.

Figure 6.6 exemplary illustrates the initially pre-populated entries when XML schema information (i.e., notebook XML schema in Listing 5.2) is provided for the

uri, prefix, and local-name partitions. The uri partition is pre-populated with the four default entries while the example schema targets the default empty string namespace URI. The prefix partition is pre-populated with the default entries according to the EXI specification [73]. Also, when an XML schema is provided the local-name partitions are pre-populated with the local name of each attribute, element, and type declared in the schema, sorted lexicographically and grouped according to the namespace URI in use. Further local-name entries are appended in the order they appear in the actual XML instance (no additional sorting is applied).
Given that the example schema uses the default empty string namespace URI it should be clear that the notebook sample assigns seven local-name entries, such as `Note` and `body`, to the empty string (i.e., `""`) URI. Whenever local-name and/or uri information items occur again, the compact identifier is used instead. For example to indicate a local-name in the local-name partition of the empty string a 3-bit compact identifier is used.

The overall value partition (see Figure 6.7) is initially empty and grows while processing an XML instance. Attribute and Character content values of the type String are assigned to this partition and it is possible to restrict the total number of value items for memory restricted devices (see EXI option valuePartitionCapacity and valueMaxLength[10]). The figure makes use of of the previously introduced Notebook example (see Listing 5.1) and assume that all value items are represented as String, as it is the case with schema-less grammars.

Figure 6.7 illustrates that value content item strings can be indexed from two different partitions, exactly one *local* value partition and a *global* value partition. The global value partition shown on the right hand side has indices to all string value content items. In our example we have six string entries (i.e., "2007-09-12", "EXI", ...) with global indices ranging from 0 to 5. Hence a global value hit can be represented with a 3-bit compact identifier. On the left hand side of Figure 6.7 we have four local value partitions grouped according to the associated qualified name. For example to represent the compact identifier for the string "2007-09-12" that also belongs to the qualified name `date` we use a 1-bit compact identifier only. Hence, the reason for differentiating between global and local partition is to signal hits in fewer bits.

We speak about string table *hits* in general if a string matches, while *misses* signalize that a given string was not found in the table. Hence, a local value hit indicates that the associates qualified names also corresponds with the current qualified name (e.g., "Shopping List" belongs to the qualified name `subject`). Any other hit refers to a global value hit.[11]

[10] `http://www.w3.org/TR/exi-primer/#exiOptions`

[11] This section describes EXI String Tables at a conceptual level. The exact bit representation of table misses and identifiers is not presented, but is described in full in the EXI specification [73].

Figure 6.6: String Table - Entries in uri, prefix, and local-name Partitions

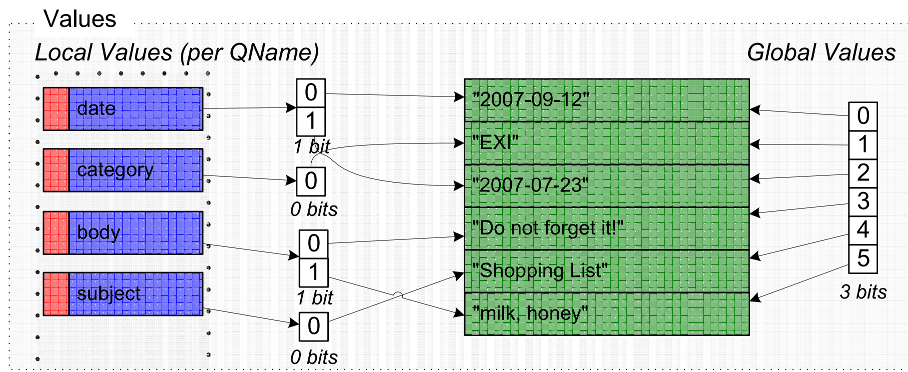


Figure 6.7: String Table - (Final) Entries in Value Partition

6.2.4 Variable-length Unsigned Integer Coding

The Unsigned Integer datatype representation in EXI supports unsigned integer numbers of theoretically arbitrary magnitude. It is represented as a sequence of octets terminated by an octet with its most significant bit set to 0 (zero). The value of the unsigned integer is stored in the least significant 7 bits of the octets as a sequence of 7-bit sequences, with the least significant byte first [73]. For example, the 8-bit sequence 00000010 is interpreted as the numeric value 2 in decimal notation and the two 8-bit sequences 10000010 00000001 are interpreted as the binary code 0000001 0000010 (removing the leading *continuation* bit) and the numeric value 130 in decimal notation.

The capability of encoding theoretically arbitrarily large numbers on the one hand and smaller numbers with fewer bits on the other hand is not new and has been used in other binary XML formats already (e.g., Fast Infoset [69] and BiM [52]). What makes it outstanding is the fact that EXI applies this concept to different areas such as the string table and datatype representations.

6.2.5 EXI Compression

EXI can use additional computational resources to achieve higher compaction. EXI compression combines knowledge of XML with a widely adopted, standard compression algorithm to achieve higher compression ratios than would be achievable by applying compression to the entire stream [73]. It multiplexes an EXI stream of heterogeneous data elements into channels of more homogeneous data elements that compress better together similar to the XMill compressors [49].

Figure 6.2 depicts an EXI Body Stream where no EXI compression is in use while Figure 6.8 shows the complementing stream when EXI compression is used. Moreover, byte-aligned data is more amenable to compression algorithms compared to unaligned representations because most compression algorithms operate

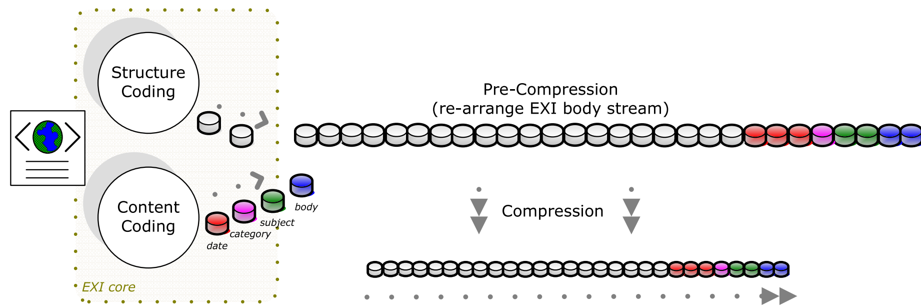


Figure 6.8: EXI Compression

on series of bytes to identify redundancies in the octets[12]. Hence, whenever EXI compression is used, event codes and content items of EXI events are encoded as aligned bytes.

XML instances and respectively also EXI instances can be treated as a combination of structure and content information. The content information can be further divided in different sections according to the context (surrounding structure as indicated by a qname). EXI treats XML instances this way and uses these implied partitions, referred to as channels, to provide blocked input to a standard compression algorithm. This grouping of similar data increases compression efficiency.

Moreover, EXI compression splits a sequence of EXI events into a number of contiguous blocks of events. Events that belong to the same block are transformed into lower entropy groups of similar values called channels, which are individually well suited for standard compression algorithms. Figure 6.8 illustrates that events inside each block are multiplexed into channels. The first channel of each block is the structure channel. The remaining channels in each block are value channels. The values of the Attribute (AT) and Character (CH) events are organized into separate channels based on the qname of the associated attribute or element (in a similar way as in XMill, see Section 6.1.1). To reduce compression overhead, smaller channels are combined before compressing them, while larger channels are compressed independently [73, 66].

Streaming EXI that makes use of EXI compression requires to buffer a block before being able to actually encode/decode the data. The first channel of the block is structural information such as elements and attributes while all other channels are value channels containing attribute values and/or character information items. The size of the block can be configured (see EXI option blockSize).

[12]Formats like BiM for instance do not provide the same performance when dealing with large documents because a bit-aligned stream cannot be compressed as well as byte-aligned data.

6.2.6 Summary

Summarising, it can be said that the Efficient XML Interchange format and its effectiveness is based on the following principles and techniques:

- Grammars describe what is most likely to occur at any given point when processing data. Moreover, grammars may be optimized using XML schema knowledge (schema-informed grammars) but do not depend on schema information at all. That said, XML instances matching XML schema definitions are represented more efficiently than XML instances that highly deviate from the expected data. In contrary to other formats, such as BiM and ASN.1, the XML instances do not need to match the XML schema to be representable in EXI.

- Event codes identify likely events with shorter paths than events that are considered less likely (similar to Huffmann tree but much simpler to compute).

- Value string tables offer local (based on QName) and global value hits to keep compact identifiers relatively small.

- Variable-length unsigned integer coding (used in structure and content value coding) reduces representation size (as in FastInfoset and BiM).

- QName-based compression (similar to XMill) works very well mainly for larger documents.

In general, most of the used techniques are not new. What is new and makes the EXI format outstanding is the ability to combine all the techniques smoothly into a single format. All principles nicely engage and do not stand by itself.

6.3 EXI Metrics

On the one hand, EXI metrics presented in the following subsections are introduced to describe properties and/or the complexity of an EXI stream. On the other hand, EXI metrics offer a tool to identify problem areas by illustrating why EXI in some cases works not as well as expected. Hence, it discloses the reason of a bad performance in regard to processing and/or compaction and highlights room for improvement.

6.3.1 EXI Content Density

Similar to XML Content Density (see Section 5.2), the EXI Content Density describes the proportion of EXI structure and the actual value content.

We differ between schema-informed and schema-less EXI Content Density depending on whether XML schema knowledge is available and should be taken into account. Different than probably expected, the special XML attributes xsi:type and xsi:nil do not account for value content but for structure given that these attributes among other things guide structure coding.

The *EXI content density* (see Equation (6.1)) bases upon *channels* as described in the EXI specification and is computed as follows:

1. Gather all value data (in Bytes) that belongs to EXI value channels[13] of the Attribute (AT) and Character (CH) events (see EXI Values).
2. The content density is the ratio of the sum in the previous step and the size of the entire EXI document in Bytes (see EXI Size).

$$\text{EXI Content Density} = \frac{\text{EXI Values}}{\text{EXI Size}} \tag{6.1}$$

The following example explains the metric by means of the example EXI document for Listing 5.1. For simplicity the EXI stream without schema information is used. The complete encoding details with a step by step approach can be found in the EXI Primer [66] document[14].

Example 6.3.1 (Content Density Computation for EXI document in Listing 5.1)**.**

1. *Value data items are*
   - *AT(date): "2007-09-12" as string*
   - *AT(category): "EXI" as string*
   - *AT(date): "2007-07-23" as string*
   - *CH: "EXI" as global value hit string*
   - *CH: "body" as string*
   - *AT(date): "2007-09-12" as local value hit string*
   - *CH: "Shopping List" as string*
   - *CH: "milk, honey" as string*

   and account for 72 Bytes.

[13] http://www.w3.org/TR/exi/#ValueChannels
[14] http://www.w3.org/TR/exi-primer/#encoding

2. *The size of the entire EXI document in Bytes is 124.*
   $EXI\ CD = \frac{72}{124} = 59\%$

In comparison with the schema-informed EXI document that has an EXI Content Density of $\frac{56}{61} = 92\%$ the difference is huge. This discrepancy can be explained by the fact that in the schema-informed case almost no structure information are to be encoded given that these information are shared between encoder and decoder. Moreover, it also discloses that improvements, if any, are to take place for the value channels.

6.3.2 EXI Efficiency

The EXI content density characterises the percentage of EXI values versus EXI structure. The ratio between EXI and XML size illustrates how well EXI is able to compress the XML instance overall. The missing link is how to detect where EXI datatype representations and/or structure can be improved. The EXI Datatype Efficiency (see Equation (6.2)) illustrates where improvements in regard to datatypes and their representations are possible while the EXI Structural Efficiency metric (see Equation (6.3)) illustrates improvements in regard to structural coding (e.g., benefit of using XML schema knowledge).

$$\text{EXI Datatype Efficiency} = 1 - \frac{EXI_{size}}{XML_{size}} \cdot \text{EXI Content Density} \tag{6.2}$$

$$\text{EXI Structural Efficiency} = 1 - \frac{EXI_{size}}{XML_{size}} \cdot (1 - \text{EXI Content Density}) \tag{6.3}$$

Both metrics highlight good performance if efficiency results get close to 1 (or respectively to 100%). Chapter 8 uses these information to identify XML/EXI test cases where an optimized datatype representation makes the most sense and presents optimized datatype representations.

The following chapters will identify what a given EXI content density and the according EXI Efficiency results reveal about EXI's efficiency regarding compression and processing time.

6.4 Discussion

In this chapter we have on the one hand introduced many binary XML candidates and on the other hand put the focus on a rather recent development, namely the

Efficient XML Interchange (EXI) format. Table 6.4 gives a summary of various candidates, its properties, and why this leads to set the focus on the EXI format and not on other formats. The selected criteria (e.g., data compression, processing efficiency, and XML compatibility) are of huge importance for applying a format to the embedded domain but also for the acceptance in the XML community. Actual performance measurements regarding data compression and processing time will be presented later in this thesis when we take a closer look at an implementation (see Chapter 7).

Compression in Table 6.4 is probably the most important criterion. When working with XML the size of the XML document is a huge drawback. Further, small and tiny documents are very common in the embedded domain. Hence, gzipped XML does not meet the data compression requirement given that for small XML documents a gzipped document becomes even larger. FastInfoset (FI) also fails due to the focus on performance rather than compression.

Processing time and efficiency is another criteria that is missed by gzipped XML given that an additional processing step (packing or unpacking) is added, compared to XML.

All listed candidates allow a small code footprint implementation that is crucial for restricted and very limited devices. Nevertheless just a few candidates can deal with XML schema definitions and may make use of the given information to provide typed data access. Further, supporting invalid XML instances according to a given XML schema (deviant data) is also very important and allows a binary XML format to smoothly integrate into the XML stack and its current applications.

XML compatibility comprises all features that XML offers, meaning that the full conformance to the XML Information set needs to be available. Many candidates miss this requirement, lacking XML's built-in features such as namespace/prefix preservation, and support for processing instructions and comments.

The capability of streaming is important for device classes that may not be able to hold all data in memory that needs to be sent, but instead produce the data on-the-fly in a streaming fashion. Further, on the receiver side interpreting data can be started before the entire stream has been received which speeds up performance also.

Royalty-free is a non-technical aspect and refers to the right to use material or intellectual property without the need to pay royalties. Any W3C Recommendation is under a royalty-free patent license, allowing anyone to implement them. BiM is the only candidate in Table 6.4 that does not meet this requirement.

EXI is the only candidate that meets all requested requirements. Hence, EXI will be used as *the* binary XML format and optimizations will be developed in embedded domain specific areas (see Part III).

	XML_{gzip}	FI	ASN.1	BiM	EXI
Compression	≠	≠	✓	✓	✓
Processing Efficiency	≠	✓	✓	✓	✓
Small Footprint	✓	✓	✓	✓	✓
Schema-informed (Typed)	≠	✓	✓	✓	✓
Schema Deviations	✓	✓	≠	≠	✓
XML Compatibility	✓	✓	≠	≠	✓
Streamability	≠	✓	✓	✓	✓
Royalty Free	✓	✓	✓	≠	✓

Table 6.4: Requirements of binary XML formats

EXI Application Range

The application of XML processing in the embedded domain ranges from powerful desktop and server machines to very limited micro-controllers. It is crucial to have a single and continuous data exchange format throughout the entire application.

XML, as also EXI, base upon the XML Information Set (XML Infoset), which describes an abstract data model of an XML document. XML and EXI in turn are specific serialization formats thereof. Hence, the expressiveness of both formats is the same and no sub- nor super-setting is required when going from one format to the other. Given EXI's XML Compatibility, the exchange format smoothly integrates into the existing XML stack and allows re-using existing techniques, libraries, and tools.

In summary it can be said that EXI provides the possibility of efficiently exchanging semi-structured data in the embedded domain. The data is serialized differently, but due to XML Infoset no mapping nor any other data conversion is required.

Part III

Optimization of XML Technologies

Chapter 7

Efficient Feature-complete EXI Processor

Chapter 6 introduces and analyzes several binary XML formats and recommends EXI as *the* semi-structured data exchange format in regard to embedded domain specific requirements. In this context, Section 6.2 introduces key features of the EXI specification [73] and describes the format on a more theoretical basis.

This chapter in turn elaborates how an actual EXI processor may be realized and illustrates efficient implementation concepts and techniques. The outcome of this work identifies EXI concepts and highlights implementation strategies. Moreover, it offers appropriate solutions for desktop/server environments and identifies factors relevant for the embedded domain. Several possible technical solutions are sketched while at least one solution has been implemented. The overall outcome is a software library, mainly meant to run on desktop environments, offering powerful XML processing with the aid of EXI technology.

A feature-complete implementation is realized with the demand to provide good processing performance in comparison to existing XML solutions. Nevertheless, the focus is to be *feature-complete* and does not take into account embedded domain requirements such as being small in regard to code footprint. The following chapters will make use of what has been learned to further optimize and apply the techniques on more restricted device classes. Hence, Chapter 8 analyzes data compression, Chapter 9 considers how to reduce code footprint, Chapter 10 will take a closer look on optimizing processing performance, and Chapter 11 considers memory-constrained querying.

The term "feature-complete processor" is generally understood as a flexible and generic processor that is aimed to support the full set of format features (in our case EXI features). Moreover, all possible combinations are meant to work

simultaneously which makes it possible to smoothly integrate in demanding and mostly powerful environments.

EXIficient[1] is the mentioned fully conforming EXI implementation developed as an open source project, implementing the W3C Efficient XML Interchange (EXI) format specification in the Java programming language. With the support of Siemens Corporate Technology, it was possible to develop the first, and so far only, open source EXI processor supporting the full set of EXI features. Siemens' application background, especially in the area of smart energy profile [7] and vehicle-to-grid [3, 4] (a system in which electric vehicles communicate with the power grid), was very beneficial for this work. Real world use cases influenced the development and led to the proposed solutions.

The reasons for launching this project were manifold. First of all, it was and still is very beneficial for the EXI working group, and the contribution as an active EXI working group member, to provide actual test data and implementation experience. Second, being able to provide actual test data and real measurements to working group members and people in the outside world is far more convincing than doing paper work only.
Beyond the proof of concept for various proposed extensions and technologies, the W3C charter explicitly required at least two interoperable implementations[2] before the EXI specification was able to move to the last W3C standardization step, becoming a W3C recommendation. EXIficient is one of three implementations that successfully passes all EXI interoperability tests and is often qualified as the de-facto standard implementation.

7.1 Implementation Techniques

XML (EXI) applications range from desktop/server applications to very restricted devices such as embedded domain microcontrollers. Due to its flexibility and its nature a fully conforming EXI processor also demands more resources. Hence, it is more applicable to desktop and/or server machines than to microcontrollers.
In contrast, a dedicated EXI processor (see Chapter 10) is meant to work for a given set of uses cases only, targets the embedded domain and runs on domain-specific microcontrollers.

Figure 7.1 illustrate requirements and dependencies of an EXI processor such as EXIficient. EXI is a grammar-based format. EXI grammars are the core component of any EXI processor and guide both, the structure and the content coding process. Grammars can be created while processing an actual XML instance (built-in XML grammars) or are derived from schema knowledge. The process of building schema-informed grammars usually requires a schema proces-

[1] `http://exificient.sourceforge.net`
[2] `http://www.w3.org/XML/EXI/implementation-report`

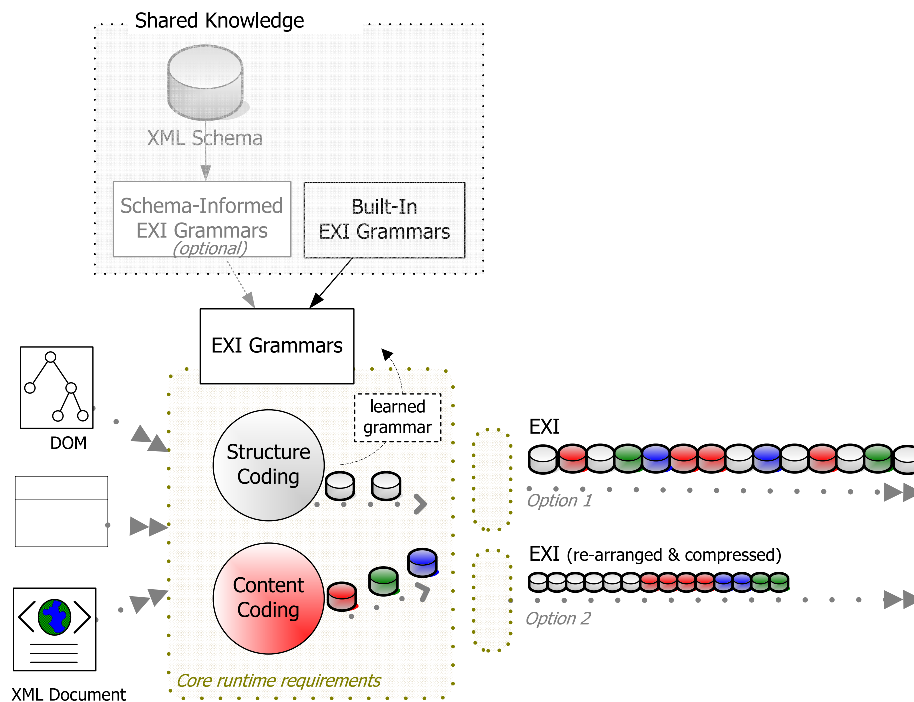


Figure 7.1: EXI Overview - Fully conforming EXI Processor

sor. EXIficient for instance uses Xerces2[3], a fully conforming XML schema 1.0 processor in the Apache Xerces family. There are many other XML schema processor candidates available in various programming languages such as Sun's XML Schema Object Model (XSOM) library in Java. The reason for selecting Xerces is of course stability and high profile but also a feature that is called "whatCanGoHere". Xerces' `XSCMValidator` offers a method called `whatCanGoHere(...)` that reports all possible elements and/or attributes at any given point in time. This is very beneficial given that it matches how EXI grammars work. At any point in time when processing an EXI stream the possibilities of future events (such as elements and attributes) differ and define the number of events and respectively influence the actuals bits and bytes of the EXI stream.

The application programming interface (API) serves as an interface between different software programs and facilitates their interaction, similar to the way the user interface facilitates interaction between humans and computers.
EXIficient supports widely used XML APIs such as Simple API for XML (SAX [12]), Document Object Model (DOM [56]), and Streaming API for XML (StAX [31]) that was meant to overcome some issues with the first two programming interfaces.

All so far mentioned APIs work on a textual basis. This means that data

[3] `http://xerces.apache.org/xerces2-j/`

is passed and reported as a string value (e.g., `"12.34"`) even if it is clear from schema knowledge that we deal with a float value for example. For XML and XML processors, this does not cause any issues given that an XML representation is represented as a sequence of characters anyway. Feeding an EXI processor with string values or requiring to report string values on decoder side demands converting the internally used typed representation (e.g., float or integer) back and forth from/to character representations. We will get back to this topic in Chapter 10 when we discuss what makes EXI processors more efficient in regard to processing cycles but also in regard to code footprint and memory consumption.

The EXI specification describes in a very detailed way how XML information is to be transformed to EXI and vice versa. Nevertheless, how this is technically realized is kept open to encourage people to come up with their best solution for their uses cases and environments.

In the following sections implementation concepts and details of the EXI processor EXIficient are discussed. Note, many techniques have been influenced by the fact that EXIficient is fully-conforming and being realized in the Java programming language. We will see more implementation approaches in the following chapters (e.g., Chapter 10) that take into account other use cases and requirements.

7.1.1 Element Context Stack

The EXI processor EXIficient uses a stack for handling XML documents. A stack entry for each element content model, when processing an EXI stream, is used. This means that when traversing (processing) an element of the XML instance tree a corresponding stack item per depth is allocated. Moreover, EXI allows representing XML instances with a valid root element (i.e., *EXI document*) and also XML instances that use any other XML element (i.e., *EXI fragment*[4]). Hence we have one initial additional stack item telling whether we deal with an XML document or XML fragment. At most *XML tree depth*+1 number of items are on the stack. Such a stack item is composed of two information tuples, the associated grammar and the qualified name of the given element (see Figure 7.2).

The stack of grammars is necessary to match XML event streams. Given that XML is not a regular language, a single grammar cannot be used to represent an entire XML event stream. The top grammar defines the current state and the likelihood of certain attributes, elements, characters and other XML information items. Every time we process a new start element a new grammar is pushed on the stack while end element events pop the stack item again.

[4] XML fragment is a general term to refer to a part of an XML document, see `http://www.w3.org/TR/exi/#key-fragmentOption`

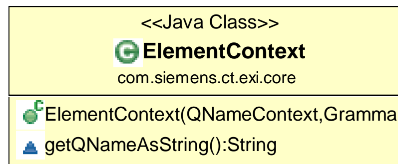


Figure 7.2: EXIficient ElementContext

As mentioned the initial stack level always consists of either the Document or Fragment grammar depending on whether we deal with an XML document or fragment. When no schema information is available, all other grammars are Built-in Element grammars. If schema information is available, we deal with a combination of either schema-informed element grammars, type grammars, and Built-in Element grammars.

The qualified name information item of the stack entry represents the qualified name of the XML element. For the sake of processing performance, we introduced the concept of a qualified name context.

7.1.2 Qualified Name Context

The overall concept of EXI are qualified names. In Section 6.2, we have explained that string tables are linked to qualified names, as well as compression channels, EXI grammars, global attributes, and elements.

In general, the implementation strategy to link grammars, string tables, and compression channels to a given qualified name separately from each other seems applicable. Another open source implementation, namely OpenEXI[5], chose to do so. This separation has the advantage that different technical concepts (e.g., string tables and compression channels) can be modified and revised in parallel. Given that no interconnection exists, revising one technique does not affect the other technique. The only commonality is the qualified name. Nevertheless, let's sketch an example EXI coding process of a string value that is also compressed using EXI compression. First of all this means that for a given string value the associated list of local value entries is retrieved (using the qualified name) to check whether we can represent the value as a local value hit. Afterwards, the same qualified name is used to retrieve the right compression channel (again using the qualified name). This processing cycle shows that splitting qualified name relevant concepts is a valid approach but it also seems to have certain shortcomings.

From the processing performance perspective, it seems more reasonable to have a single context of each qualified name. From this unique context all referring components are linked. Hence EXIficient uses this technique based on the concept of a `QNameContext`.

[5] `http://openexi.sourceforge.net/`

The `QNameContext` in Figure 7.3 first of all contains information about the qualified name in the form of local-name and namespace URI string. Further, it allows one to retrieve the actual namespace ID and local-name ID used in the EXI stream. EXI encodes namespace URIs and local-names at most once as string (e.g., "http://www.w3.org/2001/XMLSchema") and uses compact IDs (e.g., 3) any other time. This improves compression on the one hand and processing performance on the other hand. Moreover, due to IDs fast integer number comparisons are possible instead of using slower string comparisons.

Figure 7.3: EXIficient QNameContext

Figure 7.3 also depicts the relations of a qualified name context. First of all, any `QNameContext` refers to the associated URI context (`URIContext`). A `URIContext` may consist of multiple `QNameContexts` while a `QNameContext` always belongs to exactly one URI. Further, we see an optional list of simple sub-types as a self-reference and a reference to a global type grammar

(`SchemaInformedFirstStartTagGrammar`). The global type grammar is present if the associated qualified name possesses a type grammar. When schema-informed grammars are used, this is the case for all schema types (e.g., `xsd:date` and `xsd:int`) and global types defined by a user.

Additionally, we see an optional reference to a global attribute (`Attribute`) and/or a global element (`StartElement`) grammar. These information is provided if the XML schema defines such a global attribute and/or element definition.

The qualified name context is a generic concept meaning that it is applicable to schema-informed and schema-less grammars. EXI uses schema information to pre-populate qualified names and assumes them to be shared knowledge between encoder and decoder.
Qualified names stemming from XML schema (due to pre-population) are static and can be shared over multiple coding processes.
Qualified names (and respectively `QNameContexts`) stemming from an XML instance that either does not have appropriate schema information for this qualified name or does not have schema information at all are different. The qualified name context bases on a given XML instance and vanishes after processing this XML document (i.e., it cannot be shared over multiple XML instances). By definition, those qualified names do not have neither EXI simple subtypes nor global types. Further, global attribute types are not available. A global element grammar based on the evolving Built-in Element grammar is created for each qualified name that does not have a global element grammar stemming from XML schema already.

7.1.3 Grammars

EXI Grammars represent a set of productions that are available at any given point in time when processing an EXI stream. Each production in turn consists of an event that leads to a grammar again (see Figure 7.4: getNextRule() in Production Interface). Grammar productions may be composed of one, two, or three event code parts, depending on the likelihood of the event production. The more likely an event the less event code parts and the more compact the event can be represented (see *Grammar Event Codes* in Section 6.2.2).

The targeted grammar may be the same grammar as the starting grammar. In case of EndElement (EE) and EndDocument (ED) events, there is no following grammar. In case of EndDocument, the stream has been successfully processed.

While processing EXI streams we have several grammars on the stack matching the nesting XML elements and sub-elements. This means that each element has a grammar counterpart. A StartElement (SE) event pushes a grammar or respectively an `ElementContext` on the stack while EndElement (EE) events pop the top grammar from the stack again.

In addition to events that do have data associated with it (e.g., StartDocument, EndDocument, StartElement, EndElement, Comments, Processing Instruc-

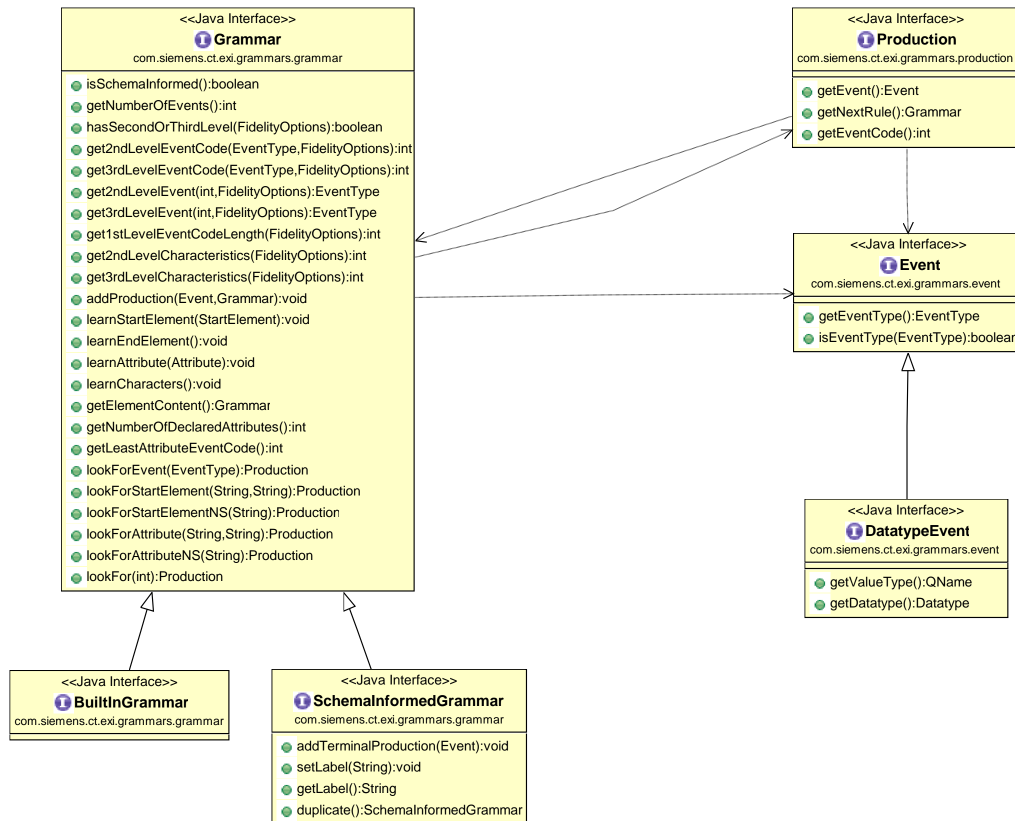


Figure 7.4: EXIficient Grammars

tion, . . .) we also know events that deliver type information (e.g., Characters and Attribute events, see Interface `DatatypeEvents` in Figure 7.4). Type information is used to represent the data of attributes and characters more efficiently. With the type information knowledge that "123" is an unsigned integer value, the data can be represented with 1 Byte in EXI instead of using 3 Bytes for the actual string characters.

7.1.4 String Table

String tables are used in memory-constrained areas allowing a compact representation of repeated string values. Re-occurring string values are represented using an associated compact identifier rather than encoding the string literally again. In EXI, when a string value is found in the string table (i.e., a string table hit) the value is encoded using a compact identifier. Only if a string value is not found in the associated table (i.e., a string table miss) the string is encoded as String and a new compact identifier is introduced.

EXI uses string tables for the following four information items:

- uri
- prefix
- local-name
- value

The following string table characteristics apply to *value* strings that belong to attribute and character values only. *Uri*, *prefix*, and *local-name* string tables work slightly different. However, *value* string tables use partitions based on the context (qualified name) in which the string occurs. Let us re-use the string table example that has been introduced in Chapter 6. For convenience, the figure (see Figure 7.5) has been included a second time.

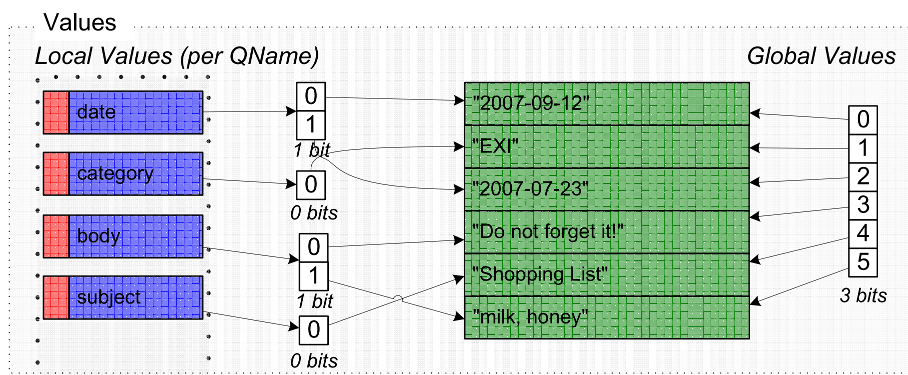


Figure 7.5: String Table Implementation Details – Entries in Value Partition

One approach to check whether a string value can be represented as a local-value or a global value hit is to look up the given local value partition (according to the qualified name) and global value partition just like the arrows in Figure 7.5 indicate. For example, check first the local-value partition and if not successful afterwards the global value partition. This also means that on encoder side each string may require two table look-ups, which can turn out to be very costly.

The nice property of EXI string tables is that according to the EXI specification [73] a given string can be at most in one local-value partition. This leads to another more optimized string table look-up approach that has been implemented in EXIfcient. The idea is depicted in Figure 7.6. One needs to keep an *overall* list of string values. Each string value points to the associated global value ID and the associated local value ID. If the string has been found int the *overall* list it is a hit, otherwise a miss. In the case of a hit, another check is required that tells whether the qualified name is the same (local-value hit) or whether the qualified name does not match (global-value hit). The local name check is done again using IDs instead of comparing strings. This approach can reduce the number of string

table look-ups by half given that at most one string table look-up per string is necessary.

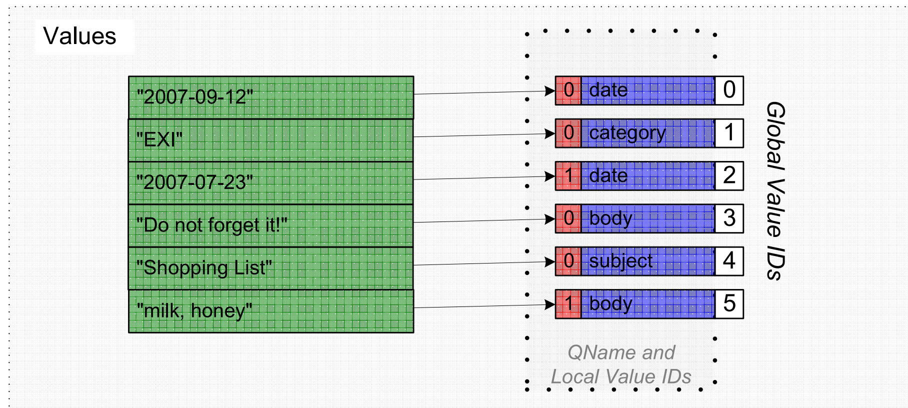


Figure 7.6: String Table Implementation Details – Optimized Value Partition

We herewith conclude the technical description of the EXI processor EXIficient by having introduced the most important concepts and optimization techniques such as qualified names, grammars, and the string table. Subsequently performance measurements in regard to data compression and processing are discussed.

7.2 Results and Discussion

This section presents the benefits of the EXI format compared to XML and gzipped XML. Tests have been run over the EXI Working Group's Measurement Test Framework[6], which contains 88 test documents from 20 test groups. The test data ranges from 99 Bytes up to 70 MegaBytes in size. Moreover, we deal with XML documents ranging from XML Content Density of nearly 0% up to almost 100% (see more detailed test data description in Section 5.3).

EXIficient - Hard Facts and Dependencies

The version of EXIficient that has been used for the measurements is release 0.9. The runnable JAR version of EXIficient (exificient.jar) has a size of about 340 kB. The currently used schema processor Xerces2 in the version 2.11.0 demands two additional JAR files with following code footprint:

- xercesImpl.jar (ca. 1350 kb)
- xml-apis.jar (ca. 216 kb)

[6] http://www.w3.org/XML/EXI/#TestingFramework

EXIficient is a Java library that runs with Java 1.5 and later versions. If no schema information is to be processed, the library does not have any further dependencies than the Java runtime environment itself. If XML schema information is to be transformed, EXIficient requires Xerces' schema processor.

7.2.1 Compression

The graph in Figure 7.7 shows EXI [EXIficient] and gzipped XML [XML.gzip] sizes as percentage of the original XML document size [XML], sorted by best compression result. Hence, a compression ratio of 50% means that the original XML document has been compressed to half of its size.

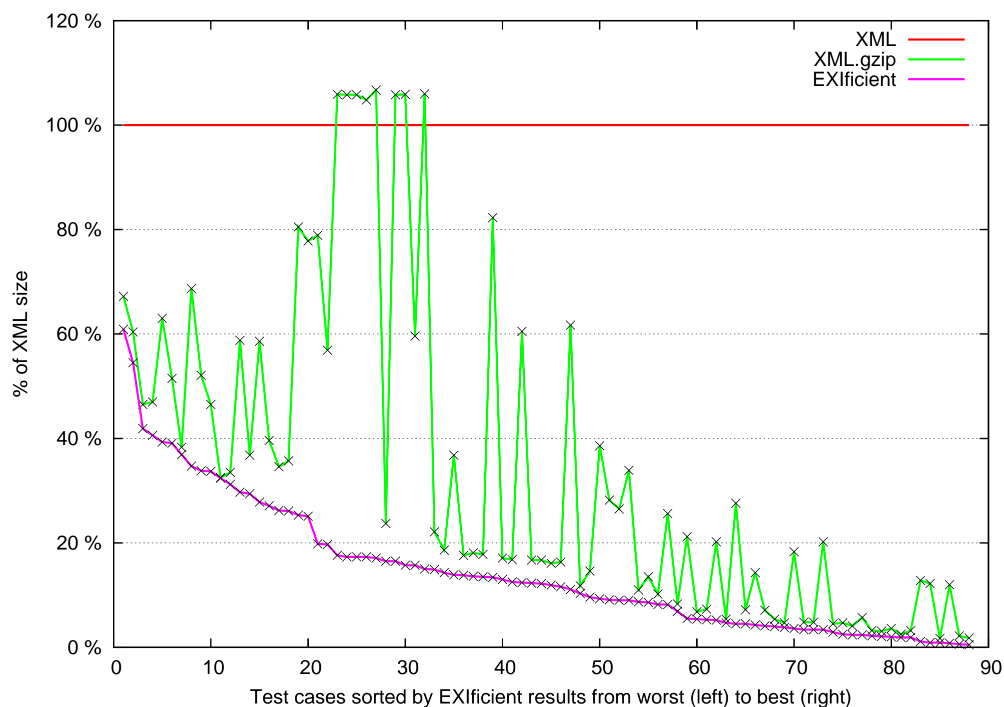


Figure 7.7: EXI Compression

Figure 7.7 highlights that EXI is consistently smaller than gzipped XML regardless of document size, document structure, or the availability of schema information. In some cases, EXI is over 10 times smaller than gzipped data. In addition, EXI works well in cases where gzip has little effect or even makes documents bigger[7], such as high volume streams of small messages typical of geolocation, financial exchange, and sensor applications. [81]

[7]Gzipping small XML documents (below ∼ 150 Bytes) can make the documents bigger due to bookkeeping overhead.

The test data is also subdivided according to the XML Content density metric introduced in Section 5.2 from high to low XML content density and from large to tiny documents.

Compression - High Content Density

High content density implies a high percentage of value data and a low percentage of XML structure. Figure 7.8 illustrates that EXI in many cases represents the data efficiently (up to 5% of the original size) due to a good datatype representation or many string table hits for heavy character biased documents. In other cases, we deal with value data that does not compress as well because we mainly deal with large non-repeating collections of character data (CLOB). Except from string tables EXI does not further analyze character data unless a restricted character set can be retrieved from XML schema patterns. The test cases with a large amount of character data that do not compress well are reconsidered in Chapter 8, where an optimized datatype representation is introduced and performance benefits are shown in regard to compression performance but also in regard to processing time. Nevertheless, even in the worst case EXI compresses XML documents to about 60% of the original size and is consistently smaller than gzipped XML.

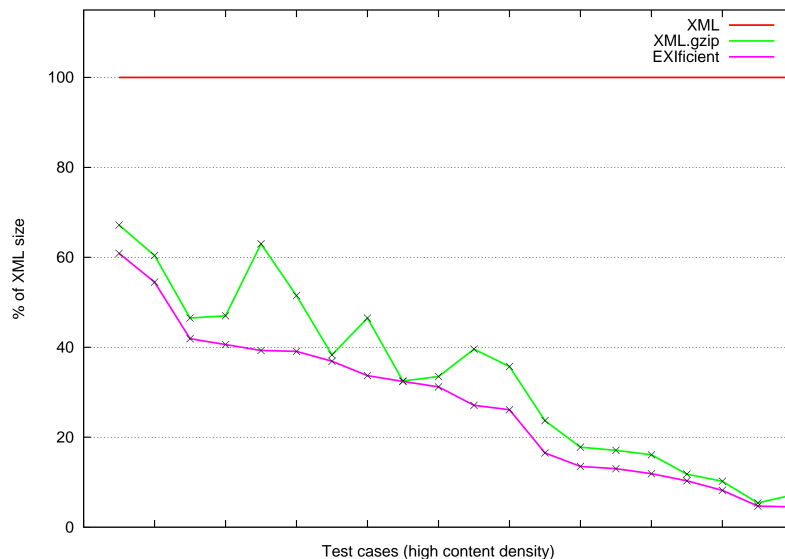


Figure 7.8: EXI Compression - High XML Content Density

Compression - Low Content Density - Large Documents

In general, large documents can be relatively well compressed using gzip. Figure 7.9 confirms this statement. All large test documents are compressed to at

most 20% of the original XML size. Nevertheless, in all cases EXI works even better. In some cases EXI is up to 13 times smaller than the gzipped counterpart. On average the compression performance of EXI is about 4% of the XML document size and hereby by a factor of 2 smaller than gzipped XML.

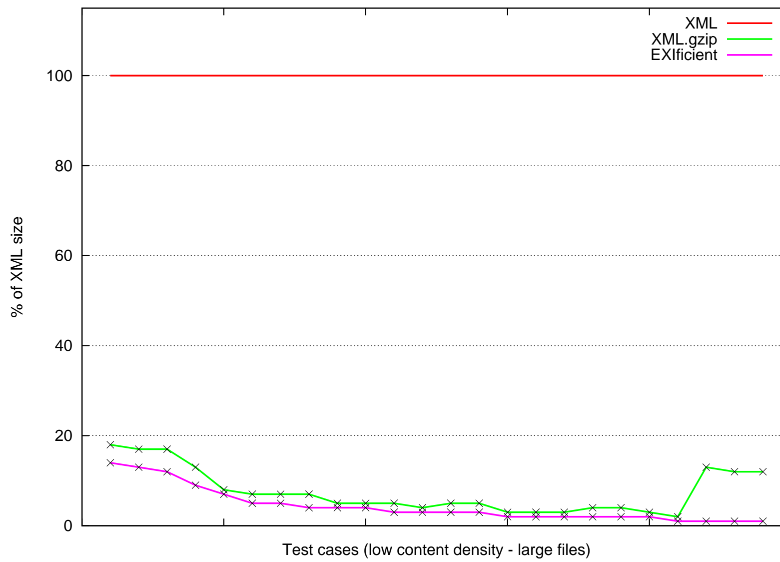


Figure 7.9: EXI Compression - Low XML Content Density - Large documents

Compression - Low Content Density - Small Documents

Figure 7.10 confirms the statement that the smaller the XML documents the larger the difference between EXI and gzipped XML becomes. The average compression performance of EXI is about 9% of the XML document size while gzipped XML accounts for 21%.

Compression - Low Content Density - Tiny Documents

Figure 7.11 approves that for tiny documents gzip does not work very well. In many cases gzipped XML documents become larger than the actual XML document itself due to the intrinsic bookkeeping overhead. EXI works consistently well with an average compression performance of 20% of the XML document size. Gzip on average accounts for 80% while at the same time adding processing cycles to compress and de-compress the actual XML data.

Summing up, we can conclude that EXI works very well over a broad range of XML test documents. In all test cases, EXI-compressed XML files turned out

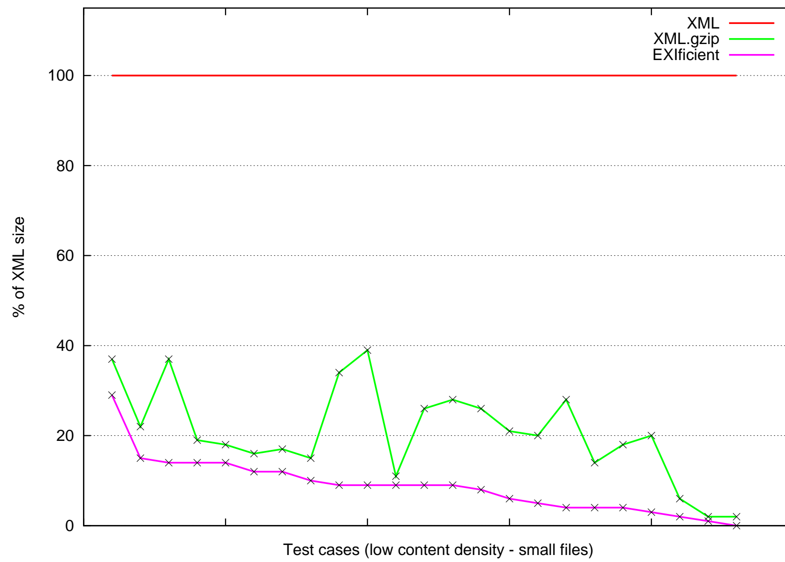


Figure 7.10: EXI Compression - Low XML Content Density - Small documents

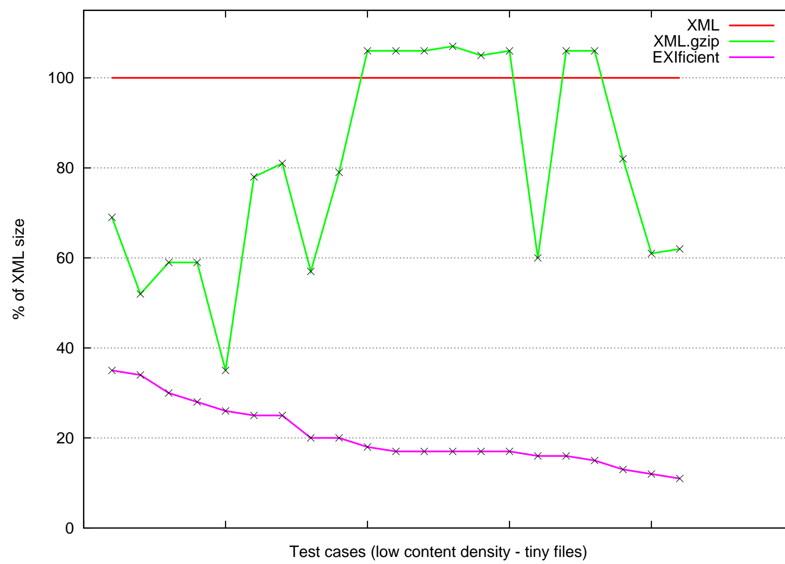


Figure 7.11: EXI Compression - Low XML Content Density - Tiny documents

to be smaller than gzipped XML files. It stands out that especially for small and tiny XML documents the difference in size becomes evident.

Given that compression size is just one aspect of a good binary XML format we subsequently analyze the processing time of EXI compared to XML.

7.2.2 Processing Efficiency

The processing efficiency tests were run using the EXI Working Group's framework[8] test data and test methodology on a Windows XP machine with an Intel Core Duo T2300 / 1.66 GHz (mobile) Processor and 2.0 Gbytes of RAM.

The following graphs (see Figures 7.12 and 7.13) illustrate the encoding (i.e., serializing) and decoding (i.e., parsing) time of EXIficient and OpenEXI without EXI compression for each test case as a percentage of XML JAXP[9] time. So, for example, a measurement of 50% means twice faster parsing, a measurement of 25% means four times faster parsing than parsing XML.

The measurements use the SAX API, meaning that all data needs to be reported as characters even if the XML schema would indicate that a given value is a float value or an integer number. This is done in order to show the benefit of using EXI where previously XML has been used. XML values are characters in all situations while EXI internally works with typed data for integers or float values. Hence users can expect an additional performance speed-up when typed APIs can be used instead of the traditional text-based APIs. The following chapters will show numbers that highlight which speed-up can be expected with the right programming interface.

In order to have a good comparison the measurements also include another EXI implementation, namely OpenEXI version 0.0198.

The measurements show processing results running encoding and decoding on the same machine. No network link has been used. A real world EXI application is generally *network-bound*, meaning that the dominant factor is the speed of the network link. Hence the data size that needs to be transmitted is of huge interest. In our profiling, on purpose, we chose not to integrate any data-link measurements. Depending on the use case, the link is very different (in regard to bandwidth and load). A GPRS link or a heavily used Wi-Fi link is slowing down data exchange. Hence the depicted processing measurements constitute the worst case scenario. Processing time is assumed to be *processing-bound* instead of *network-bound*. Moreover, the data exchange uses textual APIs such as SAX instead of a data binding layer. This is done in order to show the least performance speed-up one may expect by using EXI technology.

[8] The testing framework of the EXI working group is based on the Japex micro-benchmark framework, see `https://java.net/projects/japex`.

[9] The Java API for XML Processing (JAXP) is Java's built-in processing API and was developed under the Java Community Process JSR 5 and JSR 63.

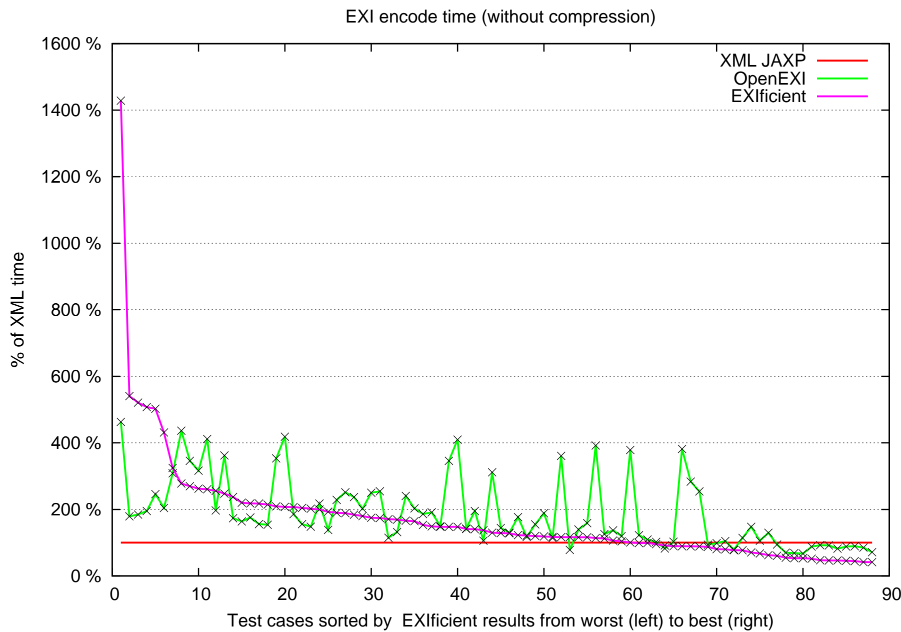


Figure 7.12: EXI Processing Time - Encoding

Figure 7.12 shows processing measurements for encoding. It is expected that transforming data from an XML instance to EXI is not as fast as decoding the data again. Given that XML data may deviate from XML schema information demands parsing text to datatype representations before actually encoding values properly. Further, in many cases the textual XML representation is very flexible and accounts for most of the transformation time to represent EXI values properly (e.g., the xsd:decimal values "1.23" and "1.230" are mapped to the same typed EXI datatype representation).
This leads to the situation that on average both EXI implementations are slower when serializing data than the XML counterpart. JAXP's XML implementation essentially writes the received characters of the SAX interface mostly as is to the disk, without any further processing. Nevertheless in cases where EXIficient manages to reduce the amount of data to be written significantly enough it also manages to be faster than JAXP.

This situation very much improves for the decoder side. Figure 7.13 depicts that the average decoding time of EXIficient was more than 3 times smaller than the average decoding time of XML. The best case was over 35 times faster. Moreover, EXIficient is in most cases also faster than OpenEXI. A detailed analysis according to the XML content density groups shows more details.

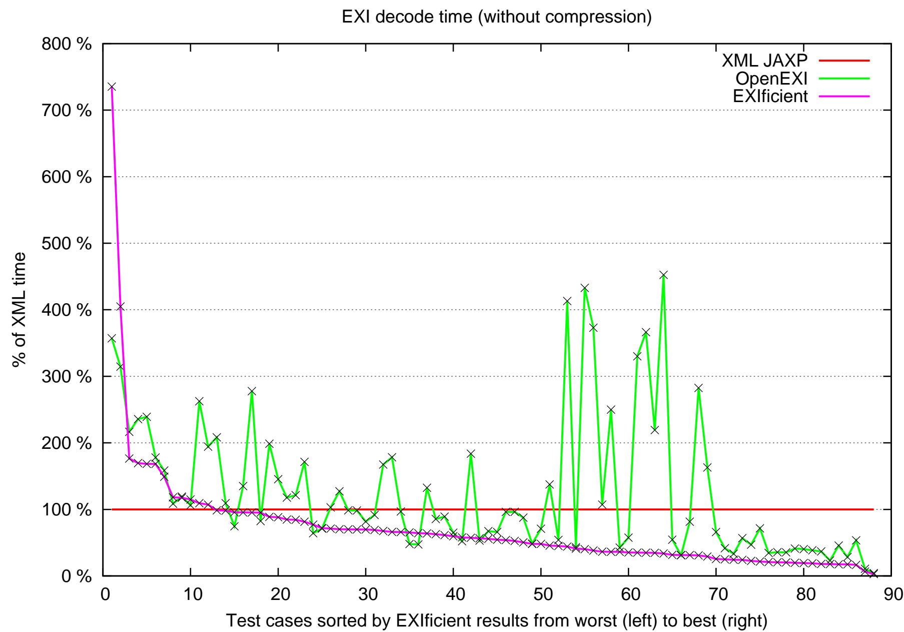


Figure 7.13: EXI Processing Time - Decoding

Decoding Time - High Content Density

Figure 7.14 depicts that both EXI implementations do not perform well on high content density data. One performance benefit of EXI processors over XML processors is that almost no XML tags have to be written or read respectively. Given that in the group of high content density the percentage of XML tags is very small the performance gain is also less evident.

Further, XML processors such as JAXP have been highly optimized while EXI is a relatively new techniques that will offer faster implementation using typed APIs over the next years.

OpenEXI is on average even 19% slower than XML processing. EXIficient is about 20% faster than XML. The best case of EXIficient is more than three times faster than processing plain XML.

Decoding Time - Low Content Density - Large Documents

Figure 7.15 depicts that both EXI implementations are working roughly at the same speed. On average OpenEXI decodes almost 3 times faster than JAXP while EXIficient decodes almost 4 times faster.

The best three cases of EXIficient are performing 6, 15, and 36 times faster than XML.

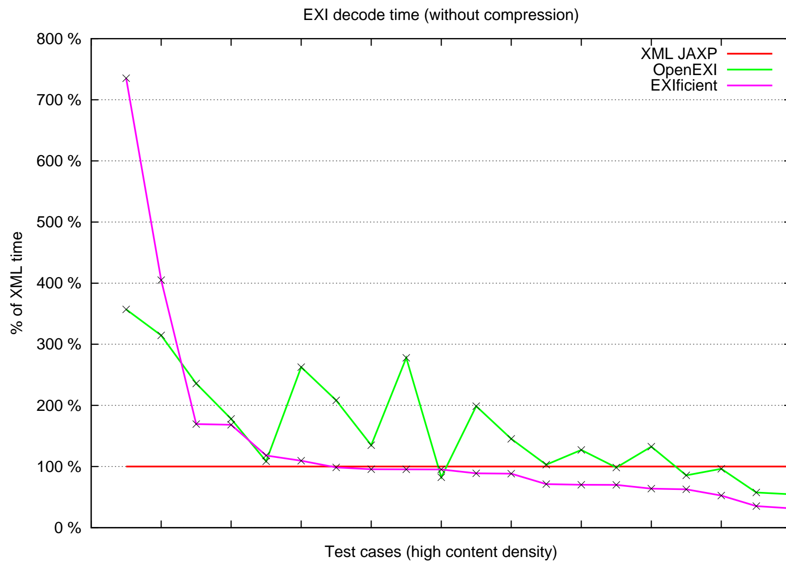


Figure 7.14: EXI Decoding - High XML Content Density

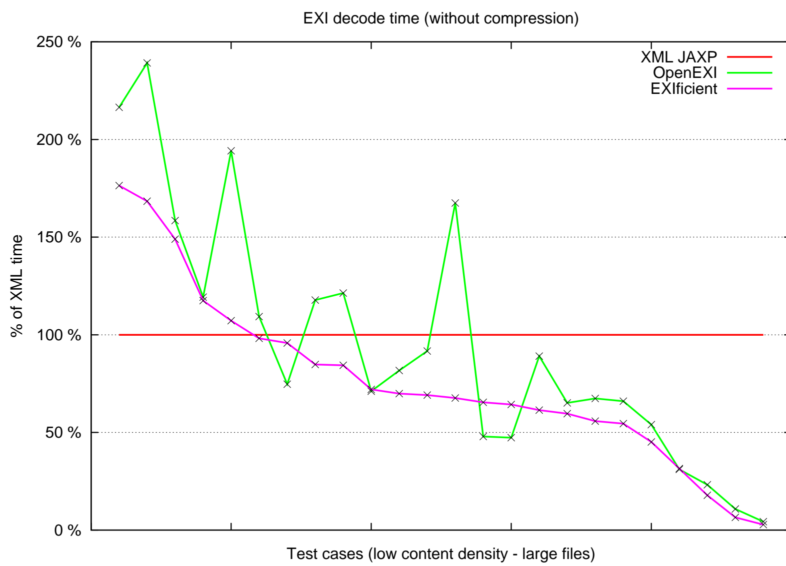


Figure 7.15: EXI Decoding - Low XML Content Density - Large documents

Decoding Time - Low Content Density - Small Documents

Figure 7.16 shows, in contrast to the previous figures, that EXIficient works significantly faster for smaller documents than OpenEXI. OpenEXI is on average about 114% of XML time while EXIficient manages to reach a value of 45%. This makes a difference in speed of more than two.

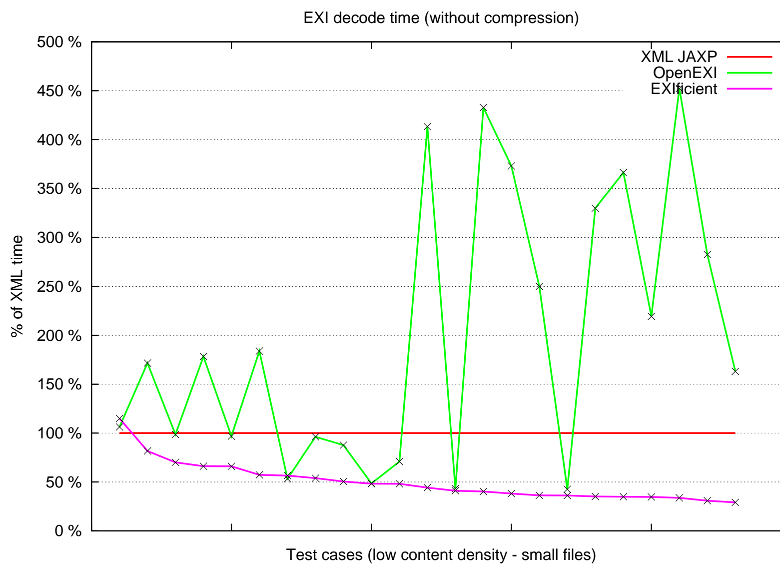


Figure 7.16: EXI Decoding - Low XML Content Density - Small documents

Decoding Time - Low Content Density - Tiny Documents

Figure 7.17 demonstrates again that EXI processors seem to work reasonably well on tiny documents. OpenEXI performs well compared to XML being just once slower than XML. The average speed gain of OpenEXI is a factor of 2.1.

Nevertheless, EXIficient performs even better and beats XML in all test cases. The speed-up ranges from 1.3 times faster than XML up to 6 times faster. This makes on average a speed-up for EXIficient of about 4.2 times over XML.

7.2.3 Optimization Outlook

We will see in the next chapters that EXI processing performance can be improved by using the right *typed* programming interface (API). Moreover, other new techniques in combination with EXI can be very beneficial in regard to processing time, compression ratio, small footprint, and memory usage. The focus for further investigations was preset to the embedded domain where processing is crucial and efficient data exchange is of huge importance.

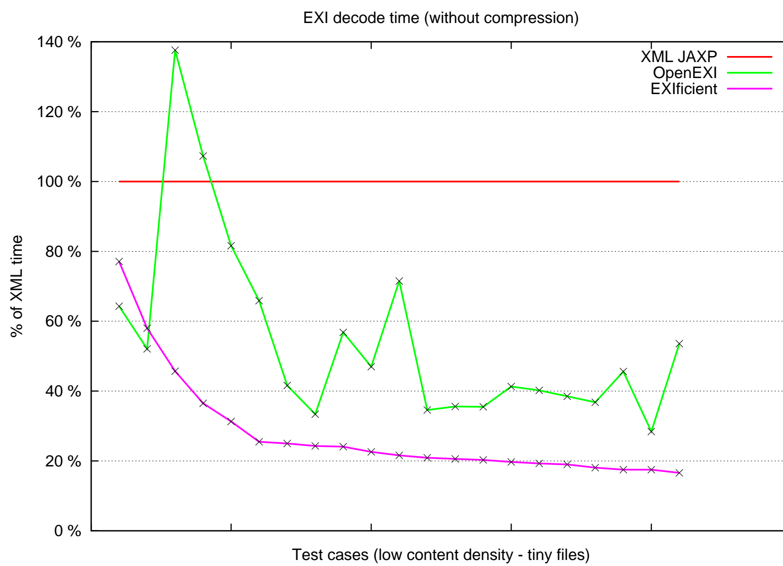


Figure 7.17: EXI Decoding - Low XML Content Density - Tiny documents

Chapter 8

Optimized Datatype Representation

An EXI stream, more concrete the EXI Body stream that carries the actual information, can be described as the combination of structural data (e.g., XML tag names) and the XML values (e.g., "123") in form of XML character information items[1] and attribute information items[2].

The EXI specification defines a set of datatype representations, called built-in EXI datatype representations, that prescribe how value content items (e.g., attribute values or element characters) are to be represented. The format disposes of eleven built-in datatypes (e.g., Binary, Boolean and Integer) and EXI processors map XML schema built-in datatypes (see Figure 8.1[3]) to exactly one default EXI datatype representation (see Table 8.1). Hence, this default mapping defines how content items are to be encoded.

The performance of the EXI default representations is usually sufficient. Compaction measurements presented in Section 7.2.1 confirm this statement. Nevertheless, in some cases an improved or also a different representation is required and the EXI specification does provide means to fulfil these requirements. EXI offers a feature called *Datatype Representation Map* [73]. A Datatype Representation Map may provide alternate built-in EXI datatype representations or user-defined datatype representations for any simple XML schema datatype.

This chapter presents one important use case where an alternate representation, different from the default EXI mapping, seems feasible. Moreover it introduces a very flexible technique that shows how user-defined datatype representa-

[1] see `http://www.w3.org/TR/xml-infoset/#infoitem.character` [79]
[2] see `http://www.w3.org/TR/xml-infoset/#infoitem.attribute` [79]
[3] XML built-in datatypes, `http://www.w3.org/TR/xmlschema-2/#built-in-datatypes`

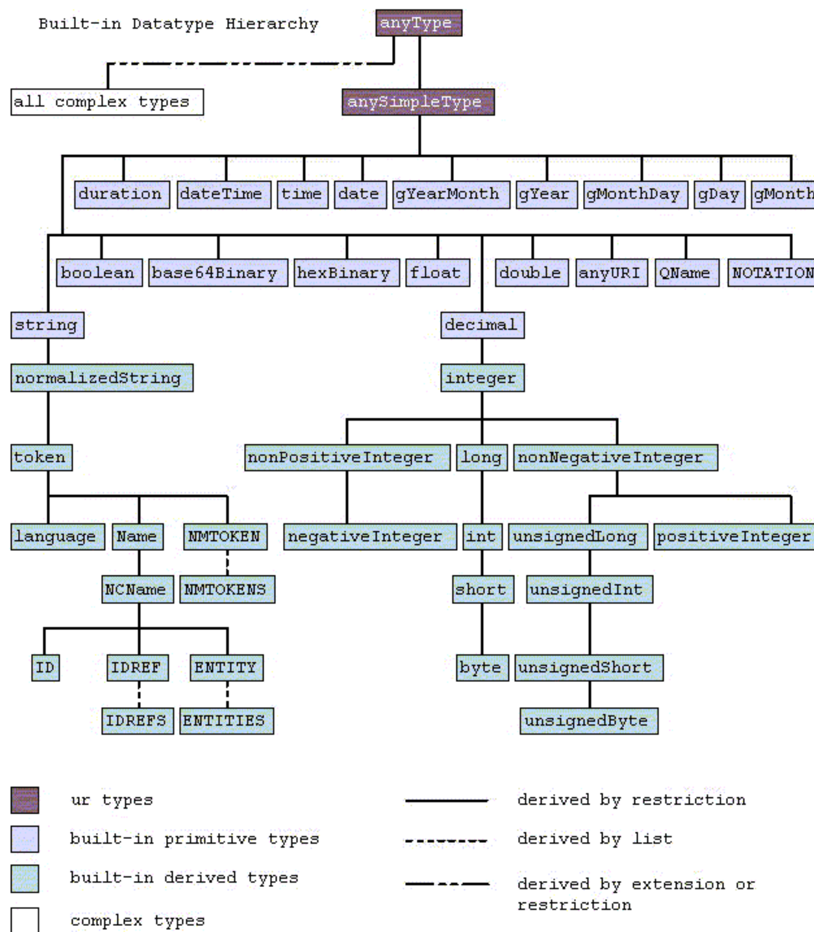


Figure 8.1: Hierarchy of XML Schema Datatypes [67]

tions can be automatically created. Emphasis is put on how such representations can be exchanged in a standardized fashion based on a well-known and widely-used notation technique.

8.1 Application Area

One field of application for EXI are Rich Internet Applications (RIAs) [65]. RIAs are web applications that tend to have similar features and alike functionality of traditional desktop applications. Traditional web applications are server centered, meaning that most of the work and the actual processing is done on servers and clients only display static content in the form of (X)HTML. To reduce server load, avoid slow interactions, and accelerate user feedback, RIA applications nowadays move processing from the server side to the clients. Client processing time is therefore of tremendous interest [44].

Built-in EXI Datatype Representation	XML Schema Datatypes
Binary	xsd:base64Binary xsd:hexBinary
Boolean	xsd:boolean
Date-Time	xsd:dateTime xsd:time xsd:date xsd:gYearMonth xsd:gYear xsd:gMonthDay xsd:gDay xsd:gMonth
Decimal	xsd:decimal
Float	xsd:float xsd:double
String	xsd:string xsd:anySimpleType and all types derived by union
Integer	xsd:integer
n-bit Unsigned Integer	Not associated with any datatype directly, but used by Integer datatype representation for some bounded integers
Unsigned Integer	Not associated with any datatype directly, but used by Integer datatype representation for unsigned integers
List	All types derived by list, including xsd:NMTOKENS, xsd:IDREFS and xsd:ENTITIES
QName	xsi:type attribute values when Preserve.lexicalValues option value is false

Table 8.1: Built-in EXI datatype representations

In many concepts such as AJAX, XML is used as the data interchange format. Microsoft's Silverlight[4], SMIL (SVG) [30], and MPEG LASeR [5, 24] are also XML-based formats. In nowadays web, data mashups combine data from more than one source and their visualization is distributed to a variety of embedded devices. Thus the focus lies on rich media applications for small and restricted devices with limited bandwidth where the use of XML is desired, but entails disadvantages or is simply infeasible. XML data is in general highly redundant. The redundancy affects application efficiency through higher storage, data transmission, and processing costs. EXI's applicability to rich media applications in general is of interest. SVG is chosen as a concrete example.

8.1.1 Scalable Vector Graphics

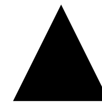
Scalable Vector Graphics (SVG) [30] is an XML specification and format description for two-dimensional vector graphics. Rich Internet Applications, multimedia presentations such as SMIL, and application frameworks such as Silverlight are built on top of it. SVGTiny or Mobile [14] are meant for less powerful devices and provide a subset of SVG's features.

SVG images and their behaviour are defined in XML text files. The SVG specification [30] defines 14 important functional areas such as basic shapes, text, and colors. One of the most important functional areas are *paths*.

SVG paths are compound shape outlines drawn with curved or straight lines and can be filled in or outlined. The XML attribute `d` builds the main information item of the element `path`. It expresses for example in a compact textual coding (see Figure 8.2), that the virtual pen is initially moved to the coordinate pair (X,Y) denoted by `M`[5]. L precedes a subsequent point to which a line should be drawn. Z closes the path.

```
<path d="M 250 150 L 150 350 L 350
    350 Z" />
```

(a) Textual SVG Path



(b) Rendered SVG image

Figure 8.2: Example of an SVG path

SVG files tend to be verbose and hardly processable on constrained end devices. Additional generic compression techniques such as gzip (known as "SVGZ") are not suitable in restricted domains due to the increased need for computational resources. In addition, if an SVG document is to be compressed as a whole,

[4]Website about Silverlight Architecture: `www.silverlight.net`

[5]Note: All of the commands can also be expressed with lowercase letters. Capital letters means absolutely positioned, lower cases means relatively positioned.

progressive rendering is partly lost since the compressed document needs to be de-compressed before being passed to an SVG viewer.

Moreover, SVG's XML documents mostly consist of only a few clob attributes and/or elements such as the path `d` attribute. The structure hereby is part of the attribute content and not structured via XML itself. An SVG engine needs to extract those tokens (e.g., coordinates) such as the ones shown in Figure 8.2 before rendering the graphic.

8.1.2 Applying EXI to Rich Web Applications

EXI is a compact representation for XML intended to simultaneously optimize performance and the utilization of computational resources. Measurements to confirm EXI's applicability in the RIA domain, based on SVG test documents, were elaborated. To facilitate reproducibility, the same SVG test set as by the EXI working group (see [81, Section 4.1.2]) has been used.

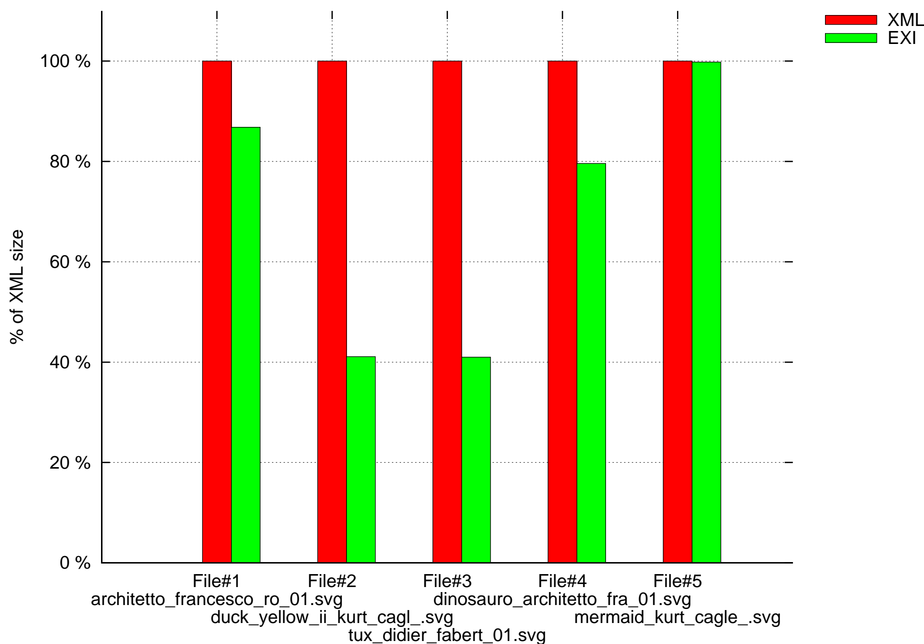


Figure 8.3: EXI Compression of SVG Data

The test data consists of a collection of five SVG files ranging from 1 kB up to 181 kB. Figure 8.3 depicts that for three out of five RIA test cases EXI's built-in datatype representation does not work as well as throughout various other areas (see performance numbers in Section 7.2.1). Further the compression ratio fluctuates from test case to test case. The compressed size compared to XML ranges from about 41% up to almost no compression at all.

Test Case	XML [Bytes]	EXI Compression	EXI Datatype Efficiency Ratio
File#1 architetto_francesco_ro_01.svg	18570	86.8%	14.2%
File#2 duck_yellow_ii_kurt_cagl_.svg	2173	41.1%	77.0%
File#3 tux_didier_fabert_01.svg	896	41.0%	82.4%
File#4 dinosauro_architetto_fra_01.svg	23229	79.6%	24.0%
File#5 mermaid_kurt_cagle_.svg	181570	99.8%	0.3%

Table 8.2: EXI Datatype Efficiency Ratio for SVG Test Documents

At the first glance the compression results are against expectation. What surprises is that EXI in some cases cannot compress the documents at all. This raises the question why this happens and what causes these problems. Section 6.3 introduced EXI metrics that turn out to be very beneficial for identifying the issue. We subsequently focus on the EXI Datatype Efficiency ratio numbers (Equation 6.2) that are shown in Table 8.2. It turns out that the compression and the Datatype Efficiency Ratio is inversely correlated. The lower the percentage of *EXI Datatype Efficiency* the less good the EXI compression. Hence this data sustains the observation that the bad compression performance mostly relates to how EXI represents the actual data or values.

When doing a more detailed analysis, some interesting observations can be made that highlight why File #2 and #3 achieve much better compaction results than the others. Figure 8.4 spreads the compaction results into structure, values without the attribute `d`, `d` only, and insignificant XML whitespaces. The implication is that the attribute `d` is hardly compressible and in those cases where it takes a very high percentage (Files #1, #4, #5), EXI compression efficiency suffers.

We conclude that EXI achieves very good results for highly structured XML data while test cases with almost no structure (e.g., some SVG files) achieve less good results. Moreover, we can also draw very informative conclusions from measurements that have been co-published with the EXI working group in [81, 9]. EXI processing is generally very fast, especially the decoding process is many times faster than conventional XML parsing. Moreover, another interesting point arises from these measurements and shows that EXI has also a strong correlation between EXI compression efficiency and processing time [9]. Hence, reducing the size of the EXI stream also reduces processing time.

The grammar for path data `d` is given by a BNF (Backus Normal Form or Backus-Naur Form), a notation technique for context-free grammars (see Listing 1

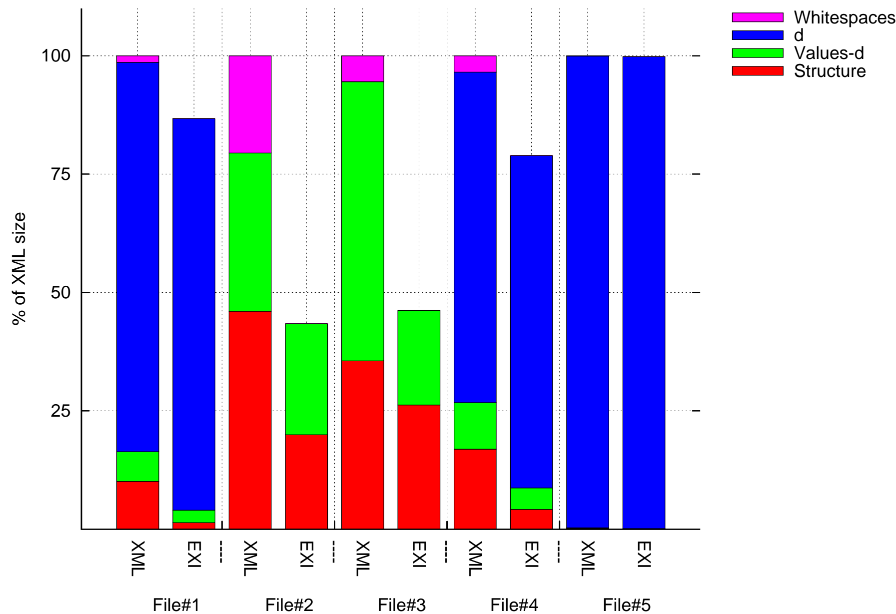


Figure 8.4: SVG Data Distribution for XML and EXI Representation

in Appendix). The following sections create the basis for introducing the concept of an optimized datatype representation for EXI values.

8.2 Formal Grammar

A formal grammar (sometimes simply called a grammar) is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context.

A formal grammar is of the form $G = (V, \Sigma, S, P)$ where $V = \Sigma \cup N$ is a vocabulary consisting of a finite alphabet Σ and a set of nonterminals N. The symbol $S \in N$ is the start symbol and P is a set of production rules with a left-hand and a right-hand side. On the left side of any rule must be at least one nonterminal.

A formal grammar defines (or generates) a formal language, which is a (usually infinite) set of finite-length sequences of symbols (e.g., strings) that may be constructed by applying production rules to another sequence of symbols, which initially contains just the start symbol.

8.2.1 Chomsky Hierarchy

When Noam Chomsky first formalized generative grammars in 1956 [15], he classified them into types now known as the Chomsky hierarchy. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages.

The Chomsky hierarchy consists of four levels and each level or type is determined by the form of the production rules in P [22, 37].
Type-0 grammars (unrestricted grammars) include all formal grammars.
Type-1 grammars (context-sensitive grammars) have rules of the form $\alpha A\gamma \rightarrow \alpha\beta\gamma$. A is a nonterminal while α, β and γ are terminals and nonterminals. The string γ must be nonempty whereas α and β may be empty.
Type-2 grammars (context-free grammars) consist of production rules that are restricted to exactly one nonterminal on the left side and arbitrary terminals and nonterminals on the right side.
Type-3 grammars (regular grammars) further restrict their production rules to a single terminal on the right-hand, possibly followed or preceded (but not both in the same grammar) by a single nonterminal.

By definition Type-0, Type-1, Type-2, and Type-3 grammars define a strict hierarchy of formal languages (e.g., Type-3 grammar is a special Type-2 grammar etc.). EXI grammars and the Backus-Naur Form (BNF) are context-free grammars. Regular expressions (abbreviated regex or regexp) are regular grammars. In summary, it can be said that all previously mentioned grammars are at least Type-2. Hence, hereinafter we deal with the expressive power of Type-2 grammars.

8.2.2 Backus-Naur Form

BNF (Backus Normal Form or Backus-Naur Form) is a notation technique [8] for context-free grammars (Chomsky level Type-2), often used to describe the syntax of languages used in computing.

BNF Example - Social Security Number

As an example, consider one possible BNF for the U.S. Social Security number (SSN)[6].

Listing 8.1: BNF for Social Security Number

```
<SSN> ::= <area-number> '-' <group-number> '-' <serial-number>
<area-number> ::= <number>
```

[6] http://www.socialsecurity.gov/history/ssn/geocard.html

```
<group-number> ::= <number>
<serial-number> ::= <number>
<number> ::= <digit> | <digit> <number>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

- The first set of three digits is called the Area Number.
- The second set of two digits is called the Group Number.
- The final set of four digits is the Serial Number.

A SSN is a nine-digit number issued to U.S. citizens, permanent residents, and temporary (working) residents and has three parts separated by hyphens in the format "AAA-GG-SSSS". The BNF in Listing 8.1 depicts a simple BNF not dealing with the number of digits in the sense that for example the area number is required to have exactly three digits.

BNF Variants

Many extensions and variants of the original BNF notation are used; generally either for the sake of simplicity and succinctness or to adapt it to a specific application. Some variants are exactly defined, including Extended Backus-Naur Form (EBNF) [1] and Augmented Backus-Naur Form (ABNF) [21]. Common features of many variants are the use of repetition operators such as * and +, or the bracket "[ ]" notation to express optionality.

EBNF Example - xsd:date

Let us investigate a more sophisticated BNF, describing the `xsd:date` datatype modeled after the calendar dates of ISO (International Organization for Standardization) 8601. Its value space is the set of Gregorian calendar dates and its lexical space is the ISO 8601 extended format with an optional time zone[7]:

`[-]CCYY-MM-DD[Z|(+|-)hh:mm]`

Valid values include: `2001-10-26`, `2001-10-26+02:00`, `2001-10-26Z`, `2001-10-26+00:00` or `-2001-10-26` and a possible EBNF for this XML schema datatype is the following Listing 8.2, which is used throughout this chapter to illustrate the concept of an optimized datatype representation.

Listing 8.2: EBNF for xsd:date

```
xsdDate = year , '-' , month , '-' , day , [timezone];
year = [negative] , digit , {digit};
```

[7] To specify a time zone, you can either enter a date in UTC time by adding a "Z" behind the date or you can specify an offset from the UTC time by adding a positive or negative time behind the date.

```
month = digit , {digit};
day = digit , {digit};
timezone = 'Z' | ( '+' | '-' ) , hour , ':' , minute;
hour = digit , {digit};
minute = digit , {digit};
negative = '-';
sign = '+' | '-';
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```

8.2.3 Parser Observations

The term parser or parsing is the process of analysing a string of symbols (e.g., the date value `2001-10-26` into its components year, month, and day). An LL parser is a top-down parser for a subset of context-free grammars known as the LL grammars. The LL parser parses the input from *l*eft to right and constructs a *l*eftmost derivation. An LL parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence.

In the interests of simplification and availability we focus on LL(1) parsers in order to parse BNF grammars[8]. LL(1) parsers recognize languages that have LL(1) grammars, which are a special case of context-free grammars. LL(1) parsers cannot recognize all context-free languages. The LL(1) languages are exactly those recognized by deterministic pushdown automata restricted to a single state [43].

Anyhow, in reality this issue usually is not critical. LL(1) grammars are very popular given that grammars with a high value of k have traditionally been considered difficult to parse.

The textual *splitting* of a string of symbols (e.g., `2001-10-26`) requires several steps and optimizations. The nonterminals `year`, `month`, `day` have to be identified to describe an integral number instead of an arbitrary selection of digits. Obvious and fixed terminals, such as the hyphens (e.g., between year and date), have been pruned (see Listing 2 in appendix). Section 8.3 will investigate in full detail how this can be done automatically.

8.2.4 XML Schema and EXI Grammars

XML schema as well as EXI grammars are grammars of Chomsky level Type-2. Moreover, all productions in a normalized EXI grammar contain exactly one terminal symbol and at most one non-terminal symbol on the right-hand side. This is a restricted form of Greibach normal form [34, 73].

That said, a parser by means of the introduced `xsd:date` BNF transforms a possible value `2001-10-26` to the following XML snippet.

[8] If the BNF is of an LL(k) grammar an LL(k) parser needs to be used or the according conflicts have to be resolved that makes the BNF again an LL(1) grammar.

```
<xsdDate>
    <year>2001</year>
    <month>10</month>
    <day>26</day>
</xsdDate>
```

8.3 Concept of an Optimized Datatype Representation

In Rich Internet Application environments that use SVG as their representation form, the attribute `d` sometimes accounts for almost 100% of the entire size (XML as well as EXI, see Figure 8.4) and an optimized representation for this specific information item is desired.

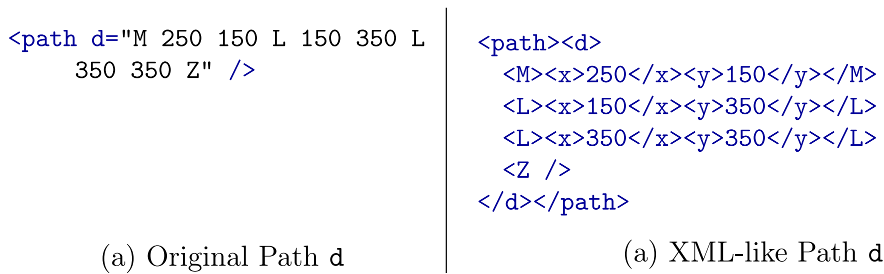
EXI usually leverages XML schema patterns (e.g., lowercase letters only) and uses restricted character sets to represent the content more efficiently. In the case of SVG paths, the attribute `d` is restricted by a Backus-Naur Form (BNF) (see [30, 8. Paths]) and is not expressed by XML schema characteristics. Hence, EXI processors need to make use of the previously described feature Datatype Representation Map and specific representations to encode information items more efficiently.

8.3.1 Datatype Representation Generation

A first straightforward approach to develop a *hand-optimized* representation for the information item `d` is maybe feasible, but surely not desirable. This manual process of analysing the complex structure of a BNF and the associated overhead may not be worthwhile. Moreover, two different developers may come up with two different representation for the same BNF. Hence, a nonproprietary solution that is specified and developed once and is flexible enough to satisfy future needs is far more interesting and useful for the XML community.

The optimized datatype representation approach, or as we used to call it, the "EXI-BNF" approach, basically leverages any available BNF or regular expression to optimize EXI datatype representation. The outcome is an user-defined datatype representation that may extend our freely available EXI implementation (see Chapter 7). The actual work is done in two consecutive steps.

In the first step, a parser splits the path information string (e.g., `M 250 150 L 150 350 L 350 350 Z`) into its symbols and produces tokens for the according path data commands such as moveto (M) and lineto (L). Secondly the extracted information tokens of `d` are transformed to an XML document or respectively to an XML fragment. Figure 8.5 depicts the idea of splitting the content and restructuring the tokens XML-like. Coding of this XML fragment does not require additional code since the same EXI library can be (re-)used for encoding and decoding.



(a) Original Path `d` (a) XML-like Path `d`

Figure 8.5: EXI Representation Idea

8.3.2 BNF Extraction Process

We differentiate between six different terms when analyzing an (E)BNF, namely *choice*, *option*, *group*, *repetition*, *terminal* and *nonTerminal*. To ease XML schema mapping we differentiate between a *named* nonTerminal and a *anonymous* nonTerminal. A BNF builds subsequently a tree composed of these terms.

choice A choice represents a choice between multiple sub-terms (e.g., `'+' |'-'`).

option An option describes an optional portion (e.g., `[negative]`).

group A group term groups terms together (e.g., `( '+'| '-' )`).

repetition A repetition depicts a repetitional part (e.g., `{digit}`).

terminal A terminal depicts the end of a term (e.g., `'0'`).

nonTerminal A nonTerminal depicts that more terms are following (e.g., hour in `hour = digit , digit`).

The process of mapping (E)BNF terms to to XML schema constructs is automatable and straightforward.

choice maps to an XML schema choice element (`<xsd:choice>`).

option makes use of the XML schema occurrence indicator (`minOccurs="0"`).

group maps to an XML schema sequence element (`<xsd:sequence>`).

repetition makes use of the XML schema occurrence indicators (`minOccurs="0"` and `maxOccurs="unbounded"`).

terminal maps to a named XML schema element constrained by an XML schema simple type.

nonTerminal maps to a named element while an anonymous nonTerminal becomes an anonymous complex type.

For example, the results of the (E)BNF Listing 8.2 is the following list of *named* nonTerminals which subsequently have additional sub-terms.

xsdDate = year , '-' , month , '-' , day , [timezone]

This BNF rule for xsdDate translated into a sequence of a nonTerminal year, a terminal '-', a nonTerminal month, a terminal '-', a nonTerminal day and an optional timezone indicated by minOccurs="0".

```
<xsd:element name="xsdDate">
        <xsd:complexType>
                <xs:sequence>
                        <!-- year = [negative] , digit , {digit} -->
                        <xsd:element ref="year"/>
                        <!-- '-' -->
                        <xsd:element name="terminal1">
                                <xsd:simpleType>
                                        <xsd:restriction base="xsd:string">
                                                <xsd:enumeration value="-"/>
                                        </xsd:restriction>
                                </xsd:simpleType>
                        </xsd:element>
                        <!-- month = digit , {digit} -->
                        <xsd:element ref="month"/>
                        <!-- '-' -->
                        <xsd:element name="terminal3">
                                <xsd:simpleType>
                                        <xsd:restriction base="xs:string">
                                                <xsd:enumeration value="-"/>
                                        </xsd:restriction>
                                </xsd:simpleType>
                        </xsd:element>
                        <!-- day = digit , {digit} -->
                        <xsd:element ref="day"/>
                        <!-- [ timezone ] -->
                        <xsd:sequence minOccurs="0">
                                <!-- timezone = 'Z' | ( '+' | '-' ) , hour ,
                                     ':' , minute -->
                                <xs:element ref="timezone"/>
                        </xsd:sequence>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Note that an EXI processor completely removes enumeration values with a single enumerated value in XML schema. That said, `terminal1` and `terminal3` are not encoded in the stream given the fact that `"-"` is the only option. Moreover, when transcoding the EXI stream back to XML the terminal is reproduced automatically again.

year = [negative] , digit , {digit}

The year concept could be transformed into a leading negative sign followed by digits. The result is a signed number. We implemented a solution that either detects such constructs or can be forced to do so by the special EBNF sequence `? ... ?` as proposed by ISO/IEC 14977 [1] (e.g., `? REPRESENT_AS <datatype> ?`). The <datatype> part can be replaced by any known XML datatype such as `xsd:short` or so.

```
<xsd:element name="year" type="xsd:short" />
```

The year concept can be re-used by referring to it as follows (see, for example, xsdDate BNF rule).

```
<xsd:element ref="year"/>
```

month = day = hour = minute = digit , {digit}

In a similar fashion to year, month, date, hour, and minute can be handled.

```
<xsd:element name="month" type="xsd:unsignedByte" />
<xsd:element name="day" type="xsd:unsignedByte" />

<xsd:element name="hour" type="xsd:unsignedByte" />
<xsd:element name="minute" type="xsd:unsignedByte" />
```

timezone = 'Z' | ('+' | '-') , hour , ':' , minute

The `xsd:date` timezone part is more complex. It can either be "Z" representing the zero-length duration timezone or an hour preceded by plus or minus and followed by a semicolon and the minute portion.

```
<xsd:element name="timezone">
        <xsd:complexType>
                <xsd:sequence>
                        <!-- 'Z' | ( '+' | '-' ) , hour , ':' , minute -->
                        <xsd:choice>
                                <!-- Anonymous = 'Z' -->
                                <xsd:element name="Anon_0">
                                        <xsd:complexType>
                                                <xsd:sequence>
```

```
                                        <!-- 'Z' -->
                                        <xsd:element name="
                                            terminal0">
                                            ...
                                        </xsd:element>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                            <!-- Anonymous = ( '+' | '-' ) , hour , ':'
                                , minute -->
                            <xsd:element name="Anon_1">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <!-- '+' | '-' -->
                                        <xsd:sequence>
                                            ...
                                        </xsd:sequence>
                                        <!-- hour = digit , {
                                            digit} -->
                                        <xsd:element ref="hour
                                            "/>
                                        <!-- ':' -->
                                        <xsd:element name="
                                            terminal2">
                                           ...
                                        </xsd:element>
                                        <!-- minute = digit ,
                                            {digit} -->
                                        <xsd:element ref="
                                            minute"/>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:choice>
                    </xsd:sequence>
                </xsd:complexType>
</xsd:element>
```

8.3.3 BNF-based Datatype Representation Approach

The following steps describe what needs to be done to use an (E)BNF for an optimized datatype representation:

1. Use or write an appropriate (E)BNF if not already present.
   The BNF describes the information items that are present and structures

the string representation. If desired specific encodings can be used indicated by the BNF construct `(? REPRESENT_AS <datatype> ?)`.

2. Use the sketched automatic generator that creates XML schema and respectively EXI grammars. Further, it also provide a lexical scanner that splits the string representation into its XML portions, e.g., transforms `"2001-10-26"` into

```
<xsdDate>
    <year>2001</year>
    <month>10</month>
    <day>26</day>
</xsdDate>
```

3. Any EXI processor that is able to support Datatype Representation Maps can be set to use the previously generated EXI grammars for representing the data provided by the lexical scanner. We have been selecting EXIficient as the EXI implementation.

8.4 Results and Discussion

8.4.1 Measurements and Evaluation

With the presented Optimized Datatype Representation inspired by the SVG Path `d` BNF, the EXI compression measurements with and without the optimization have been re-done (see Figure 8.6).

The automatic generator has been fed with the SVG Path BNF (see Listing 1) and produced as outcome the according XML schema (see complete Listing 3 in Appendix). Each EXI processor can use this XML schema document to create its EXI grammars as described by the EXI specification.

Figure 8.6 indicates that compared to XML, the EXI-BNF-aware solution achieves an overall data reduction of about 50% for the SVG test cases. The test documents that previously did not compress very well (see EXI File#1, #4, and #5) are almost cut to half. Apparently the EXI-BNF-aware solution does not have any effect on test documents that do not contain SVG Path `d` data at all (see File#2 and #3).

The compression results are very convincing. Moreover, based on the *standardized* BNF format, interoperable data exchange between various EXI libraries can be accomplished. This means that standardization bodies can refer to a given BNF that has to be used for various EXI datatype representations and do not need to define any further steps when applying EXI. In many cases, like also for SVG, an appropriate BNF is already available.

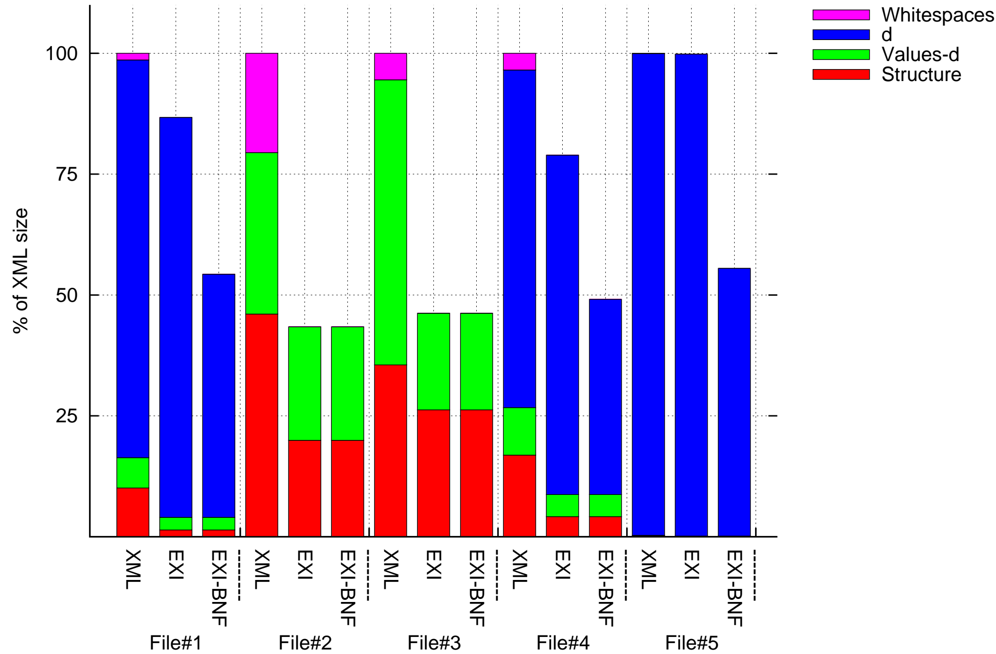


Figure 8.6: SVG Data Distribution by means of the EXI-BNF-aware solution

Further investigations in the SVG tool chain show that SVG engines such as the widely used Batik[9] viewer split information according to path types (e.g., arcs, curves, lines) before rendering the information. Our developed representation is able to deliver the information in the required separated form so that no additional parsing is required anymore (e.g., coordinates are provided as separate x/y tuples and not as the lexical string `"250 150"`).

Summarizing, we can say that our automatable EXI-BNF extension, based on regular expressions or BNFs, can be used to leverage EXIs compaction results. Moreover, by considering SVG tool chains such as Batik an improved processing efficiency can be expected. To do so, no manual tuning nor hand-optimized codec development is needed.

8.4.2 Entropy Coding

In the telecommunication sector, entropy coding for efficient data exchange is widely used. This means that a test data set and its frequency are taken into account when creating an *efficient* datatype representation. Hence, further analyses of sample data streams are done to tune data exchange. One example is

[9] `http://xmlgraphics.apache.org/batik/`

LASeR [5, 24], the MPEG standard for rich media services. LASeR also uses a binary XML encoding format (BiM) for compressing SVG documents and creates specific codecs for given data values. These codecs are influenced by entropy. This entropy estimation, or in the XML field the estimation of which item is the most frequent one, offers benefits and drawbacks.

Thomas Kurz proved in his diploma thesis [44] that entropy estimation further optimizes the EXI-BNF approach presented in this thesis. More likely events get encoded with fewer bits while unlikely events get punished. Nevertheless, a good entropy estimation demands good and relevant test documents to be available in advance so that its entropy can be taken into account. Further, the capability of automatically generating the alternate datatype representation is hardly possible anymore. Moreover, exchanging the BNF between different parties is also not sufficient given that different test-data sets create different datatype representations.

This leads to the advice that additionally using entropy coding techniques in heterogenous environments demands carefulness. Entropy estimation has its rights to exist, most likely in closed environments. On the contrary, solely using the EXI-BNF concept presented in this chapter is flexible, permissively, automatable, and hence more feasible for heterogenous networks.

Chapter 9

EXI Grammar Representation

EXI grammars play a major role when processing EXI streams. We differentiate between grammars that are built and expanded during runtime (schema-less grammars) and grammars that are either shared or generated once (schema-informed grammars). The latter type, schema-informed grammars, are based on XML schema knowledge and EXI processors exchange a so called *schemaId* to identify which set of schema documents has been used to encode XML, or respectively EXI information.

As stated already in previous chapters, the EXI specification defines how XML schema definitions are to be transformed so that EXI grammars can be built in an interoperable way. Nonetheless, the EXI specification explicitly does not define how or in which way an EXI processor is meant to exchange schema-informed EXI grammars. The reason for doing so is mostly flexibility and applicability. A proposed solution of the EXI working group may work for many use-cases but is unlikely to fit all needs. Hence, it is up to each application or application field to choose the right strategy.

This chapter will present a very flexible and efficient way to exchange schema-informed EXI grammars targeting application fields that demand supporting larger sets of XML schema files. Further, it focuses mainly on keeping the requirements in regard to processing and amount of data exchange at the very minimum. Moreover, the runtime requirements such as external library support and the associated code footprint is of particular importance.

9.1 XML Schema Exchange

Due to the flexibility of the EXI format specification in regard to how to exchange XML schema knowledge, it is likely that different EXI solutions come up

with different strategies for exchanging XML schema information or respectively schema-informed EXI grammars. The probably most obvious, very flexible, and generic approach can be summarized as follows: an EXI processor generates EXI grammars based on the exchanged *schemaId* itself. Hence, the given *schemaId* is interpreted as an URI or any other resource identifier that is accessible to the EXI processor. Consequently, the set of associated XML schema files are parsed at runtime. Doing so requires that both ends, sender and receiver, dispose of the same XML schema documents and are able to access them if required. As one prominent example, EXIficient (see Chapter 7), the feature-complete EXI processor, works exactly that way. Note, that if the EXI stream has been produced without taking into account schema information there is no need for exchanging schema-informed EXI grammars.

Another valid solution for targeting dedicated EXI services can be that the EXI processor supports EXI grammars of one XML schema (or a restricted set of schemas) only, which probably may be already pre-built into the processor. Chapter 10 will further analyze this idea.

Both so far sketched solutions are valid and follow the EXI format specification. Nevertheless, the use of these techniques is limited to a very restricted set of use cases.

9.1.1 Requirements

Many application fields, and also some of our server-aimed use cases[1], target relatively powerful device classes, demanding the flexibility of being capable to process a large set and a variety of XML schema files. Often required schema information is not available at the receiving side. Instead the schema information is to be exchanged *on the fly*. Moreover, the communication link between sender and receiver is limited. This is probably the main reason for using a binary XML format, such as EXI, in the first place. Hence one main requirement is that the communication link does not get flooded with exchanging XML schema information when the primary goal is to efficiently exchange XML messages.

Further, the EXI processor is required to be able to build EXI grammars at runtime. Following the straight-forward approach of exchanging XML schema files demands in addition to the EXI processor itself an XML schema processor such as Xerces-J. Xerces-J accounts for more than 1500 kBytes executable code while the runtime code for the EXI processor EXIficient is only about 370 kBytes. Given that EXI is the desired technology and the XML schema parser is just the required extra, this code footprint relation is not acceptable. Moreover, the code footprint of the overall deployed software needs to be much smaller. Hence, the

[1] The server part of embedded domain applications (and the associated EXI processor) often must be able to support a variety of different clients (or XML/EXI documents conforming various XML schema documents).

main requirement is to make the XML schema parser library obsolete due to its dominance in regard to code footprint.

Moreover, creating grammars from XML schema documents is time consuming also and the time of creating EXI grammars is crucial. Until the appropriate grammars are built, an EXI processor is not able to process the according EXI stream.

Hence, the goal must be to efficiently exchange schema knowledge over a limited communication channel without introducing any extra burden such as external libraries or spending a long time actually building the required EXI grammars. The previously mentioned problems have been identified and summarizing it can be concluded that the subsequently listed requirements are to be fulfilled for efficiently exchanging an arbitrary number of XML schema documents between different parties:

- Efficiently exchange schema knowledge or respectively EXI grammars over a limited communication link.
- Decrease time for using and/or switching to a given set of EXI grammars on a deployed system.
- Eliminate runtime requirements for external libraries such as dependencies on XML schema parser libraries (e.g., Xerces-J).

9.1.2 XML Schema Knowledge Exchange for Binary XML

Many schema-based binary XML formats (see Section 6.1) face the issue that exchanging schema knowledge efficiently is of tremendous interest. The BiM [36] format accommodated a standardized way that allows to encode XML schema documents as BiM streams. BiM makes use of an XML schema that defines the vocabulary of an XML schema document, the so-called *XML schema for schema*[2]. The XML schema for schemas defines how an XML schema looks like using the XML language itself.

Representing XML schema documents as binary XML encoded streams provides a solution for efficiently exchanging schema knowledge. Hence, XML schema documents can be efficiently serialized using the binary XML format itself. However, it does not provide a solution for the other two raised problems. An XML schema processor is still required for the deployed system, to be able to parse the XML schema definitions. In addition, this grammar building process demands a lot of time for generating the internal schema-informed structures.

The dependency on an XML schema parser is not unique to EXI. That said, there have been efforts to remove this requirement for the BiM binary XML format also. For example, Ulrich Niedermeier elaborated in his dissertation [58] a

[2]XML schema for schema, see `http://www.w3.org/2001/XMLSchema.xsd`

model he named *Bytecode* that represent XML schema information. Among other things the so-called *Bytecode* represents automaton information relevant for BiM processors as well as shared string entries. The different automaton states in the Bytecode emulate XML schema datatypes, definitions, and relevant restrictions (for more details, see [58]). However, the *Bytecode* model is not described by XML syntax. Hence, it is not representable with the binary XML format itself. Instead it follows hand-crafted rules.

9.1.3 XML Schema Knowledge Exchange for EXI

As mentioned in the beginning of this chapter, XML schema information in EXI streams is identified with a so called *schemaId* and its associated set of XML schema documents. Since XML schema documents are XML documents itself, a valid approach for EXI is to exchange schema documents as XML or more efficiently as EXI-encoded counterparts (similar to the BiM binary XML format). Nevertheless, doing so does not solve all identified requirements and a more appropriate solution needs to be developed.

Before defining how such an exchange format may look like, it must be first identified which XML schema rules are to be exchanged between different parties. As revealed in previous chapters, not all XML schema facets and restrictions are taken into account for building EXI grammars or do have an effect when building EXI grammars. Hence first of all it needs to be identified which information belongs to the required content of the exchange format.

9.2 Contents of XML Schema Exchange

The EXI format bases essentially on two pillars. First of all, XML tag names of attributes and elements are expected to be known on both sides, encoder and decoder side. Hence there is no need for exchanging those *known* qualified names (URI and local-names) within an EXI stream again and again.

Second, EXI grammars and their associated productions describe what is likely to occur at any given point in time when processing an EXI stream. We not only differ between schema-informed and built-in grammars but more fine-grained between different kinds of each (see Section 9.2.2).

9.2.1 Qualified Names

A qualified name, often referred to as a QName, defines an identifier for elements and attributes. QNames are defined as a combination of namespace URI and a local part of the QName. Often an XML prefix is used as a placeholder for the namespace URI.

In EXI schema-informed qualified names result in pre-populated string table entries. Hence each expected namespace URI and local-name gets pre-populated in the EXI string tables. Further, qualified names may have of an associated global element and/or attribute grammar, which in turn results in an EXI grammar.

9.2.2 EXI Grammars

We differ between various types of schema-informed EXI grammars. All grammar types share the same structure or template, meaning that a grammar consists of a set of productions. A production in turn is identified by an EXI event (e.g., StartElement, Attribute, Characters, EndElement) pointing to the next following grammar.

Figure 9.1 depicts EXI grammars for the running notebook XML schema example (see Listing 5.2 in Section 5.1.1). `F` states depict *FirstStartTagContent* grammars, `S` *StartTagContent* grammars, and `E` *ElementContent* grammars. Double-circled states denote final states, respectively states that dispose of End-Document (ED) or EndElement (EE) events.

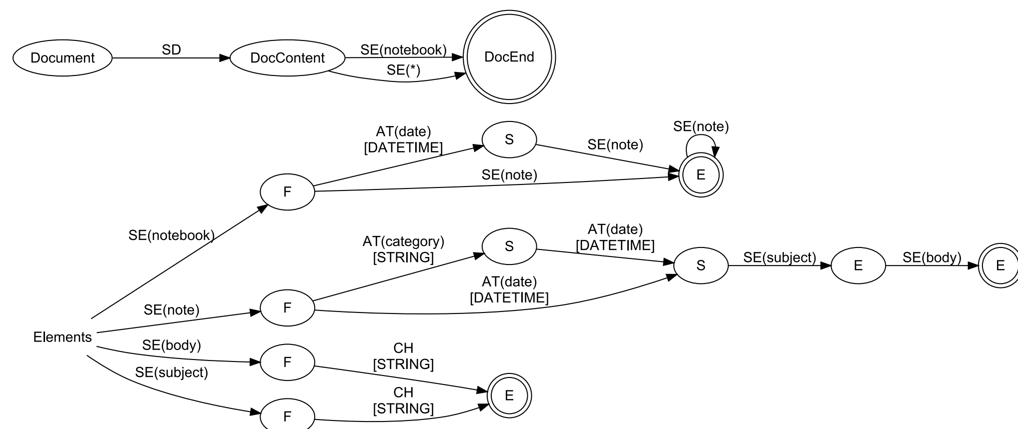


Figure 9.1: Schema-informed EXI grammars for notebook.xsd

Each EXI grammar's initial state is the *Document* grammar. There is only one possible transition (SD ... StartDocument) that moves on to the *DocContent* grammar. The XML schema for `notebook`s allows one root element only, namely `notebook`. This is denoted by the *DocContent* grammar that apart from the possibility to represent any start element SE(*), has a SE(notebook) transition only. Following a StartElement (SE) event means that first the according transition is to be taken. In this specific example, the current state moves on from the state *DocContent* to *DocEnd*. Moreover, any StartElement (SE) event pushes a new stack item on the grammar stack with the appropriate grammar.
Elements in Figure 9.1 refers to all possible start element grammars. In this case

the SE(notebook) is pushed on the stack. On the contrary, EndElement (EE) events pop one stack item again. The SE(notebook) grammar provides the event sequence for an attribute `date` and requires at least on element `note`. SE(note) in turn has of an optional attribute `category` followed by a required attribute `date`. SE(note) is composed of an element `subject` followed by an element `body`. Both elements, `subject` and `body`, are simple types denoted by the datatype string.

Schema-informed Document Grammars

The initial set of grammars when processing EXI are the document grammars which essentially represent all possible global elements that can be used to start an XML or respectively an EXI instance (see Table 9.1).
The notation that is subsequently used for grammars is as follows. On the left hand side is the grammar name (e.g., *Document*) with a number of productions indicated by a unique event (e.g., one production only with SD standing for start document). Each production points to the next grammar (e.g., DocContent) and has an event code associated to it (e.g., 0). An event code is represented by a sequence of one, two, or three parts that uniquely identifies an event (see Section 6.2.2). For example the Processing Instruction (PI) production of the DocContent grammar has a sequence of three parts (n+1).1.1 in which n stands for the number of global elements declared in the schema.

Table 9.1: Schema-informed EXI document grammars

Syntax		**Event Code**
Document :		
	SD DocContent	0
DocContent :		
	SE (G_0) DocEnd	0
	SE (G_1) DocEnd	1
	...	...
	SE (G_{n-1}) DocEnd	n-1
	SE (*) DocEnd	n
	DT DocContent	(n+1).0
	CM DocContent	(n+1).1.0
	PI DocContent	(n+1).1.1
DocEnd :		
	ED	0
	CM DocEnd	1
	PI DocEnd	1.1

Note: The variable n in the grammar above is the number of global elements G declared in the schema

In the case of schema-informed document grammars we differentiate between *Document*, *DocContent* and *DocEnd* grammars. For example the *Document* grammar has one production only. This single EXI event StartDocument (SD) with the event code 0 (zero) leads to the next grammar, namely *DocContent*.
The *DocContent* grammar, has StartElement (SE) productions for all globally available elements G. The variable n denotes the number of global elements. For example let us assume having n==3 global elements. The associated event codes for the global elements are represented by one event code part (0, 1, and 2) and the event code for SE(*) is 3. Respectively the event code for processing instructions (PI) is 4.1.1 represented by three event code parts. In this specific example the first event code part is encoded with 3 bits while the second and third event code part require one bit each. Please consider *Grammar Event Codes* in Section 6.2.2 for more details about event codes.
The *DocEnd* grammar provides the possibility to close an EXI Stream with the last possible event, EndDocument (ED), that does not have anymore a following grammar.

Schema-informed Fragment Grammars

Besides the possibility to represent valid XML documents with a single root, EXI is also capable to encode streams that represent XML fragments. XML fragments may have multiple root elements and/or represent sub-elements of a given XML instance.

Table 9.2: Schema-informed EXI fragment grammars

Syntax	**Event Code**
Fragment :	
SD FragmentContent	0
FragmentContent :	
SE (F_0) FragmentContent	0
SE (F_1) FragmentContent	1
...	...
SE (F_{n-1}) FragmentContent	n-1
SE (*) FragmentContent	n
ED	n+1
CM FragmentContent	(n+2).0
PI FragmentContent	(n+2).1

Note: The variable n in the grammar above represents the number of unique element names declared in the schema. The variables F represent these qualified names.

Table 9.2 depicts two schema-informed fragment grammars, namely *Fragment*

and *FragmentContent*. The *Fragment* grammar is the starting point for each EXI fragment while the *FragmentContent* grammar provides StartElement (SE) events for all *known* elements. Hence, *FragmentContent* differs from the *DocContent* grammar that only accounts for all global elements.

Schema-informed Element and Type Grammars

Element or type grammars describe what is likely to occur within a given schema-informed element or type. We differ between *FirstStartTagContent*, *StartTagContent*, and *ElementContent* grammar.

In contrast to *ElementContent* grammars, both *StartTagContent* grammar types may have attribute events. That said, attributes always belong to the start of an element and respectively to a *StartTagContent* grammar. Further we differentiate between the first and any other *StartTagContent* grammar. The *First-StartTagContent* grammar is special in the sense that it is the only grammar that accounts for namespace declaration (NS) and selfContained (SC)[3] events. Moreover, it is also the only grammar where specific `xsi:type` or `xsi:nil` attribute productions are available, depending on the actual XML schema document in use.

Once the grammar moves on to an *ElementContent* grammar it is not possible to represent attributes again. This matches with the XML behavior, once characters or sub-elements are processed it is not possible to encounter attributes in this element context again.

Table 9.3 shows a simplified version of one *StartTagContent* and one *ElementContent* grammar. In reality, there is one *FirstStartTagContent* grammar, followed by zero to multiple *StartTagContent* grammars, followed by zero to multiple *ElementContent* grammars. Schema-informed events are pre-populated as described by XML schema constraints. Moreover, depending on the used EXI fidelity options some EXI events (e.g., comments (CM) or processing instructions (PI)) may be also pruned. Pruning EXI events such as removing comments, if not necessary or not desired, accounts for better compression and higher processing speed.

XML Schema Particles and their Occurrences

In XML schema *particles*[4] contribute to the definition of contents models (e.g., model groups such as `<sequence>` and `<choice>`). An XML schema particle component [76] has the following properties[5]:

[3]SelfContained elements in EXI may be read independently and can be indexed for random access [73].

[4]XML schema particle component, see `http://www.w3.org/TR/xmlschema11-1/#cParticles`

[5]`http://www.w3.org/TR/xmlschema-1/#cParticles`

Table 9.3: Schema-informed EXI StartTagContent and ElementContent grammars

Syntax		**Event Code**
StartTagContent :		
	StartTagContent Event_0	0
	..	...
	StartTagContent Event_{n-1}	n-1
	EE	n.0
	AT (*) StartTagContent	n.1
	NS StartTagContent	n.2
	SC Fragment	n.3
	SE (*) ElementContent	n.4
	CH ElementContent	n.5
	ER ElementContent	n.6
	CM ElementContent	n.7.0
	PI ElementContent	n.7.1
ElementContent :		
	ElementContent Event_0	0
	..	...
	ElementContent Event_{n-1}	n-1
	EE	n
	SE (*) ElementContent	(n+1).0
	CH ElementContent	(n+1).1
	ER ElementContent	(n+1).2
	CM ElementContent	(n+1).3.0
	PI ElementContent	(n+1).3.1

Note: The variable n in the grammars above represents the number of EXI events declared in the schema.

- **minOccurs**
  A non-negative integer.

- **maxOccurs**
  Either a non-negative integer or *unbounded.*

- **term**
  One of a model group, a wildcard, or an element declaration.

When *term* is an element declaration, the number of such elements must be less than or equal to any numeric specification of *maxOccurs*; if *maxOccurs* is unbounded, then there is no upper bound on the number of such children.

Figure 9.1 depicts a rather simple and straightforward example without any XML schema particles appearing more often than once. The reason for introduc-

ing the concept of XML particles is that `maxOccurs` and `minOccurs` attributes in an XML schema document do have a strong impact on how EXI grammars are to be built. In order to highlight the effect, a sequence declaration in the XML schema for notebook.xsd is slightly changed from

```
<xs:sequence maxOccurs="unbounded">
   <xs:element name="note" type="Note"/>
</xs:sequence>
```

to

```
<xs:sequence maxOccurs="3">
   <xs:element name="note" type="Note"/>
</xs:sequence>
```

As one may have noticed the attribute `maxOccurs` has changed the value from `"unbounded"` to `"3"`. The effect of this minor modification is larger than one may expect. In respect to EXI grammars and the number of grammar states and transitions, two additional grammar states in the SE(notebook) grammar have been added and the state transitions have been slightly changed (see Figure 9.2 compared to Figure 9.1). As the reader may expect, changing the `maxOccurs` value to `"100"` adds another 97 new grammar states.

Figure 9.2: Schema-informed EXI grammars for modified notebook.xsd

At the moment this excursion is completed by drawing the attention to the fact that a large non-negative `maxOccurs` number compared to `unbounded` may impact EXI grammars a lot. However, we will come back to this issue when presenting some measurements and giving guidelines.

9.2.3 XML Schema Exchange Representation for EXI

As indicated, the proposed XML schema exchange approach focuses on a solution that works without any XML schema processor. This allows one to ship EXI processors without any XML schema parser library. Hence after removing the XML schema processor capability, the only remaining code on a device is the capability of processing EXI streams. We therefore narrow the solution space so that it does not require the device to demand extra libraries. The solution that subsequently is presented solely bases on EXI and pre-processed EXI grammars.

As pointed out in the introductory part, XML schema exchange representation defined in an XML format is very sensible. This allows one to encode the information very efficiently using EXI itself. In contrary to the binary XML format BiM, which also uses this strategy, our proposed solution goes one step further and uses a schema for *grammars* XML schema document to represent the data instead of a schema for schema approach. Hence, we exchange the necessary information in a pre-processed form so that EXI processors may use the grammars without any external libraries and without spending power and processing time to build EXI grammars from the given XML schema information.

Further, there are different possibilities how the knowledge can be exchanged. One possibility is to send the information as part of the EXI stream in the EXI header section. The knowledge can also be exchanged out of bound or in another hand-shake protocol. The presented proposal does not restrict its use to a given scenario nor does it force to be used in a given fashion. An application is free to pick the appropriate process.

The gain is, as highlighted, nearly no additional code to be able to process EXI grammars. The capability of parsing EXI is per se available and the only requirement is a built-in grammar for the schema for grammars definitions. With the capability of parsing schema for grammars definitions any XML schema or respectively any EXI grammar can be parsed and exchanged. There is no need for an additional schema parser library such as Xerces-J. Moreover these minimal requirements are applicable to powerful devices but also to smaller device classes with limited processing power.

9.2.4 Schema for Grammars

Before describing in full details the schema for grammars it seems reasonable to summarize which kind of information EXI is using from the broad spectrum XML schema is offering. Not all schema definitions or parameters lead to grammars or change grammars respectively. Further, we need to keep in mind how EXI grammars can be defined in XML so that an EXI processor can use the exchanged information without the need of spending too much power and time transforming them into internal structures.

In summary, it can be said that we differ between EXI grammars and qualified names that need to be present in the exchange format for an EXI processor. For further optimizations, it seems reasonable to provide a way to switch off unlikely features that can be switched on again if required. Doing so helps to make the representation even more concise. Consequently, the root element of the schema for grammars depicted as `exiGrammar` (see Figure 9.3) has three sub-elements, namely the optional `featureDatatypeRepresentationMap` followed by the required `qnames` and `grammars` elements.

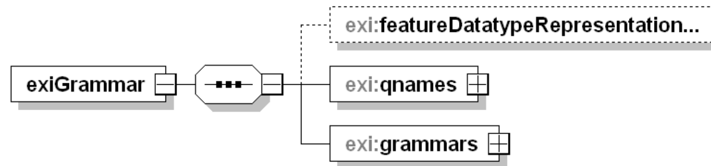


Figure 9.3: Schema for Grammars - Root element `exiGrammar`

Feature Datatype Representation Map

By default, each typed value in an EXI stream is represented using its default built-in EXI datatype representation. However, EXI processors may provide the capability to specify alternate built-in EXI datatype representations or user-defined datatype representations for specific schema datatypes. This capability is called a Datatype Representation Map [73].

Datatype Representation Map is an optional EXI feature for any conformant EXI processor and is very unlikely to be useful in many application fields. Hence the schema for grammars by default turns off the associated overhead.

That said, when the element `featureDatatypeRepresentationMap` is present it is also required to transmit a list of simple sub-types of any given simple type. In contrary, if the element `featureDatatypeRepresentationMap` is not present, simple sub-types are not exchanged as part of the XML schema exchange format. Simple sub-types are necessary to build a simple-type type hierarchy that in turn is demanded by the EXI feature Datatype Representation Map.

QNames

Figure 9.4 depicts `qnames` in EXI, the overall collection of namespace URIs and local-names. Therefore the element `qnames` contains an attribute that tells how many namespace URIs (`numberOfUris`) and qnames (`numberOfQNames`) are pre-populated due to the given XML schema information. Further, according to the number of URI, we dispose of `numberOfUris` `namespaceContext` elements, which in turn are identified by the actual URIs and the `qnameContext`. The reason why the namespace URI string is represented as any element (see `##any` in Figure 9.4)

may be surprising but becomes clear if one takes into account that EXI represents element names very efficiently. Doing so the URI can be re-used very efficiently for other qualified names. Also, for example the XML schema definitions are pre-populated in any case[6] with the URI strings "http://www.w3.org/XML/1998/namespace" or "http://www.w3.org/2001/XMLSchema-instance" that are actually never really exchanged as lexical strings.

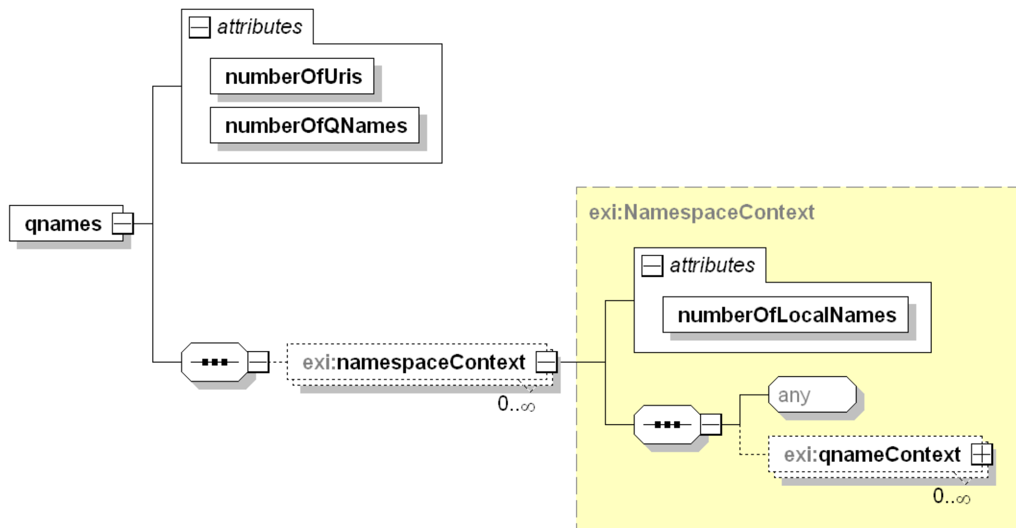


Figure 9.4: Schema for Grammars - Element `qnames`

QNameContext

The `qnameContext` (see Figure 9.5) has one required `any` element that identifies the actual qualified name URI and local-name. In addition, it informs whether the qualified name has a simple datatype (`simpleDatatype`) and a global type grammar (`globalTypeGrammarID`) and/or a global element grammar (`globalElementGrammarID`). Grammar IDs refer to EXI grammars that are available in the `grammars` element section.

Further a `globalAttributeTypeQnameID` is given if the qualified name has a global attribute and an optional list of simple sub-types is given if there are any and the feature `featureDatatypeRepresentationMap` is turned on.

XML differs between `simpleTypes` and `complexTypes`. The difference between those two kinds of types is that simple types are defined using a simple datatype such as `xs:integer` or `xs:float` while complex types may be composed of other elements, which again can be of a simple or complex type.

[6] `http://www.w3.org/TR/exi/#initialUriValues`

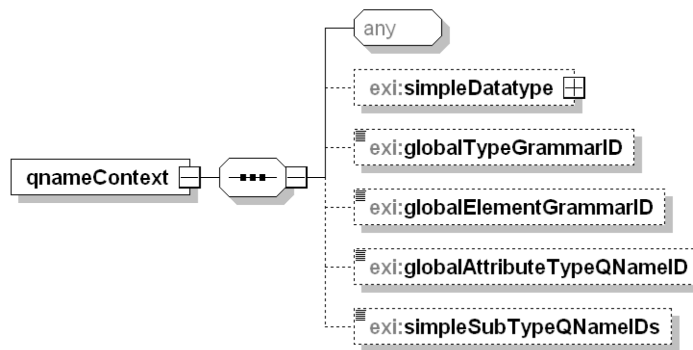


Figure 9.5: Schema for Grammars - Element `qnameContext`

EXI does also know the concept of simple datatypes that have their pre-defined representations. Nevertheless, in terms of grammars there is no difference between simple and complex types. However EXI differentiates whether a type (simple or complex) can be referenced or not. In XML terminology we speak about global types and anonymous types.

Grammars

As Figure 9.6 highlights, EXI differentiates between eight different schema-informed grammar types. According to a given *schemaId* we dispose of exactly one `document`, `docContent` and `docEnd` grammar responsible to deal with EXI documents. Further, we also have exactly one `fragment` and `fragmentContent` grammar for dealing with EXI fragments. Further we have multiple `firstStartTagContent`, `startTagContent`, and `elementContent` grammars. The attributes `numberOfFirstStartTagGrammars`, `numberOfStartTagGrammars`, and `numberOfElementGrammars` inform about the overall number of these grammar types.

Each grammar type has a unique ID that can be referenced. Hence, in theory it would not be necessary to sort the available eight grammar types in the way it is done (e.g., `document` before `docContent`). Nevertheless from the perspective of EXI event code lengths, it is very beneficial given that doing so reduces the amount of bits for identifying a given grammar type.

All eight presented grammar types base on a list of EXI productions (see `Production` in Figure 9.7). The `firstStartTagContent` grammar is different in the sense that in addition to the default grammar types it provides the possibility to define whether a schema-informed grammar is type-castable or nillable. An element is type castable if the XML schema provides sub-types to the currently used types (e.g., an element typed as `xsd:integer` is type castable given that its type can be casted to `xsd:short`). Similarly an XML schema document tells

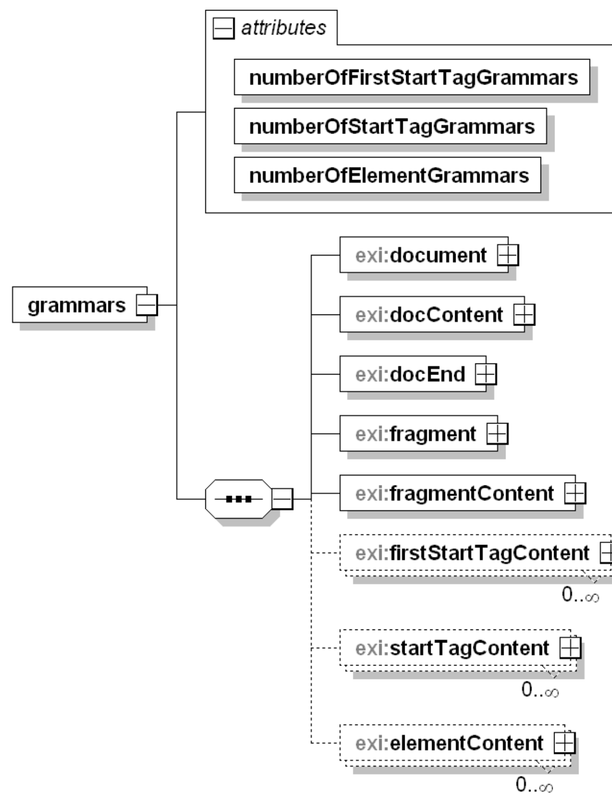


Figure 9.6: Schema for Grammars - Element `grammars`

whether an explicit null value can be assigned to the element with the attribute `nillable`.

Schema for Grammars - Summary

The presented *schema for grammars* definitions are the ones needed for representing XML schema exchange knowledge for EXI. The complete schema for grammars document listing can be found in the Appendix, Listing 4. The focus when defining the schema has been manifold. First the structure should match in-memory data structures so that it is as easy as possible to create such grammar structures. Also, the used memory and performance has been in focus and therefore many `numberOf<XYZ>` attributes are available to predict the number of entries before actually building in-memory structures. Moreover, the serialized EXI stream for XML schema exchange must be very efficient. The next section will present numbers that are about to validate these statements.

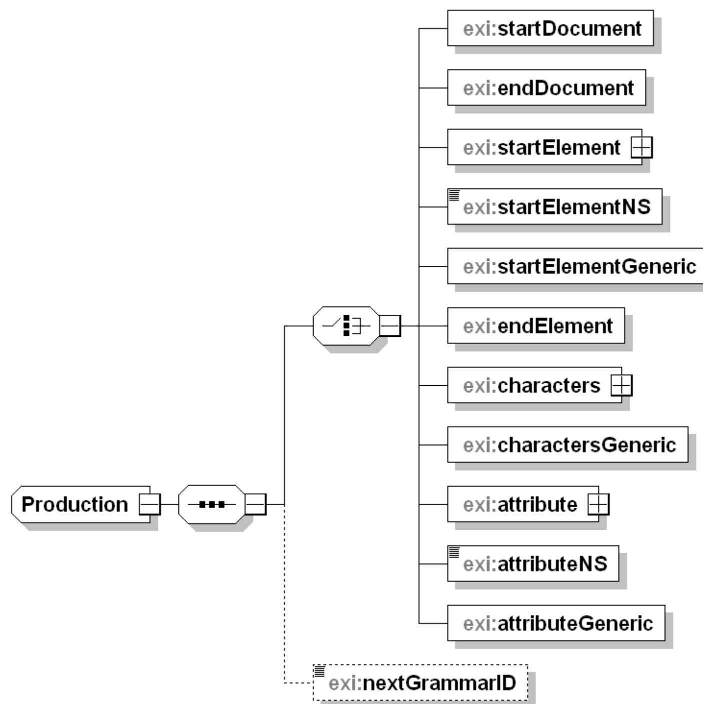


Figure 9.7: Schema for Grammars - Type Production

9.3 Results and Discussion

This section will illustrate the effectiveness of the previously proposed schema for grammars solution. Comparisons to other solutions are given even if most of the other test candidates do not fulfill the set of necessary requirements that have been identified in Section 9.1.1.

9.3.1 EXI Grammars Start-up Costs

Due to the fact that the proposed schema for grammars solution does not have any schema parser on the targeted device, an EXI implementation lacks also the so called XML built-in datatypes[7] that are omnipresent in any schema parser.

Built-in datatypes such as `xs:integer` or `xs:float` and the according grammars build the basis for any datatype-aware processing. Hence this information constitutes the *start-up costs* and is to be exchanged also. This means exchanging an empty XML schema (see listing below), which in EXI terminology stands for the availability of built-in XML schema types, has some unexpected overhead associated to it.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

[7] http://www.w3.org/TR/xmlschema-2/#built-in-datatypes

```
  <!-- built-in XML schema types only -->
</xs:schema>
```

The start-up costs for an empty XML schema following the schema for grammars rules is significant. The output for the `qnames` sections consists of the 46 built-in types. There are no global elements available. However, EXI processors have a set of shared strings and a number of different string partitions pre-populated already. The string table represents the initial entries when XML schemas are used to inform the grammars[8]. With an empty schema in total we account for 4 namespace URIs and 52 qualified name entries.

```
<exi:qnames numberOfUris="4" numberOfQNames="52">
    <!-- URI '' -->
    <exi:namespaceContext numberOfLocalNames="0">
        ...
    </exi:namespaceContext>
    <!-- URI 'http://www.w3.org/XML/1998/namespace' -->
    <exi:namespaceContext numberOfLocalNames="4">
        ...
    </exi:namespaceContext>
    <!-- URI 'http://www.w3.org/2001/XMLSchema-instance' -->
    <exi:namespaceContext numberOfLocalNames="2">
        ...
    </exi:namespaceContext>
    <!-- URI 'http://www.w3.org/2001/XMLSchema' -->
    <exi:namespaceContext numberOfLocalNames="46">
        <xs:U/>
        <!-- QNameID==6: {http://www.w3.org/2001/XMLSchema}ENTITIES
            -->
        <exi:qnameContext>
            <xs:ENTITIES/>
            <exi:simpleDatatype>
                <exi:datatypeList>
                    <exi:datatypeBuiltInType>STRING</exi:
                        datatypeBuiltInType>
                </exi:datatypeList>
            </exi:simpleDatatype>
            <exi:globalTypeGrammarID>5</exi:globalTypeGrammarID>
        </exi:qnameContext>
        <!-- QNameID==7: {http://www.w3.org/2001/XMLSchema}ENTITY -->
        <exi:qnameContext>
            <xs:ENTITY/>
            <exi:simpleDatatype>
                <exi:datatypeBuiltInType>STRING</exi:
                    datatypeBuiltInType>
            </exi:simpleDatatype>
```

[8] http://www.w3.org/TR/exi/#initialStringValue

```
        <exi:globalTypeGrammarID>6</exi:globalTypeGrammarID>
    </exi:qnameContext>
    ...
    <!-- QNameID==51: {http://www.w3.org/2001/XMLSchema}
        unsignedShort -->
    <exi:qnameContext>
        <xs:unsignedShort/>
        <exi:simpleDatatype>
            <exi:datatypeBuiltInType>UNSIGNED_INTEGER</exi:
                datatypeBuiltInType>
        </exi:simpleDatatype>
        <exi:globalTypeGrammarID>22</exi:globalTypeGrammarID>
    </exi:qnameContext>
  </exi:namespaceContext>
</exi:qnames>
```

The EXI grammars section is relatively simple, given that the only possible start element is generic that can be casted later on to a given built-in type. Nevertheless, for each simple built-in type as well as for `xsd:any` a grammar is created. Note that purely following the rules of the EXI specification may lead to duplicate EXI grammars. This is for example the case for the EXI grammars of the global type `xsd:string` and its subtype `xsd:normalizedString`. An additional optimization step removes such duplicates before creating the schema for grammars instance.

```
<exi:grammars numberOfFirstStartTagGrammars="20"
    numberOfStartTagGrammars="0"
    numberOfElementGrammars="3">
    <!-- GrammarID == 0: Document[START_DOCUMENT] -->
    <exi:document>
        <exi:production>
            <exi:startDocument/>
            <exi:nextGrammarID>1</exi:nextGrammarID>
        </exi:production>
    </exi:document>
    <!-- GrammarID == 1: DocContent[START_ELEMENT_GENERIC] -->
    <exi:docContent>
        <exi:production>
            <exi:startElementGeneric/>
            <exi:nextGrammarID>2</exi:nextGrammarID>
        </exi:production>
    </exi:docContent>
    <!-- GrammarID == 2: DocEnd[END_DOCUMENT] -->
    <exi:docEnd>
        <exi:production>
            <exi:endDocument/>
        </exi:production>
```

```
    </exi:docEnd>
    ...
    <!-- GrammarID == 27: Element[START_ELEMENT_GENERIC, END_ELEMENT,
         CHARACTERS_GENERIC[STRING]] -->
    <exi:elementContent>
        <exi:production>
            <exi:startElementGeneric/>
            <exi:nextGrammarID>27</exi:nextGrammarID>
        </exi:production>
        <exi:production>
            <exi:endElement/>
        </exi:production>
        <exi:production>
            <exi:charactersGeneric/>
            <exi:nextGrammarID>27</exi:nextGrammarID>
        </exi:production>
    </exi:elementContent>
</exi:grammars>
```

The EXI specification defines even more grammars than the ones that have been shown in the listing (see Section 9.2.2). For example for each element grammar a *nillable* element grammar is defined. This is different to the element grammar in the sense that the nillable grammar does not have character or element content events and is applied if an attribute `xsi:nil="true"` is used in an EXI stream. For compression reason, this grammar is not exchanged. Instead it can be reconstructed if needed by using the default grammar for a given element and pruning irrelevant EXI events.

Subsequently the gains and the contribution of the proposed format are discussed and evaluated according to the given requirements: exchanged size, parsing time, and code footprint.

9.3.2 Test Set

The test set has been explicitly selected to show the broad range and the effectiveness of the proposed solution (in regard to size and complexity). To do so XML schema documents from the public EXI working group testing framework[9] were selected but also three prominent and relevant standardized XML schema documents were further analyzed.

To take into account the start-up costs (see Section 9.3.1), also the *Empty* XML schema document has been tested. *Notebook* as another test document is our running example. It refers to a very simple XML schema document, which essentially is meant to show the difference from an empty schema (start-up costs) to a first real XML schema document. *Datastore* is one example that covers data-

[9]EXI Testing Framework: `http://www.w3.org/XML/EXI/#TestingFramework`

oriented XML documents of the kind that appear when XML is used to store information. The test case *GAML* covers data that is largely numeric and used in scientific applications. Web services are well covered by the test case *WSDL* that covers both, messages and other types of documents in the Web.

In order to consider real world use cases, two additional XML schema documents have been chosen. The test case *V2G* stands for the schema that is used in the ISO/IEC 15118 standardization process to specify the so-called "Vehicle 2 Grid Comunication Interface", which has selected EXI as its exchange format. *SEP2* stands for Smart Energy Profile 2.0 and is being developed to create a standard and interoperable protocol that connects smart energy devices in the home to the Smart Grid[10].

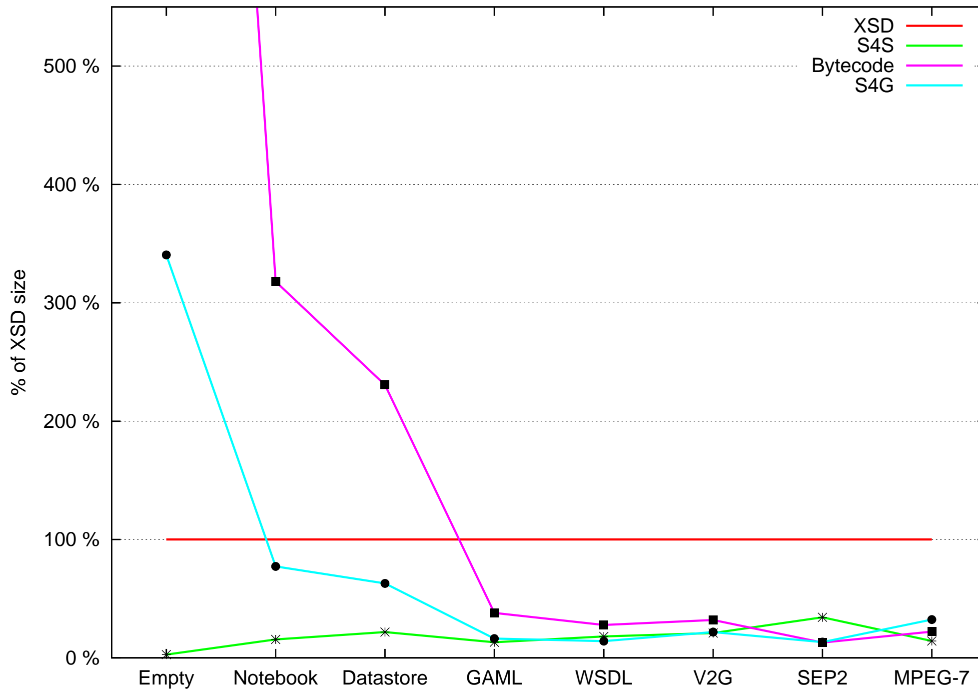
Moreover, *MPEG-7* is a multimedia content description standard and was standardized in ISO/IEC 15938. The multimedia description is associated with the content itself and allows for example fast searching. MPEG-7 uses XML to store this metadata and attaches timecodes in order to tag events.

9.3.3 Compression Measurements

One important aspect to compare the relevance of an XML schema exchange format is the amount of data that has to be exchanged. Hence this section compares the different concepts in regard to size. *XSD* stands for the XML schema language as defined by the W3C and written in plain-text XML. *S4S* stands for schema for schemas and uses the schema for schemas definitions to apply EXI coding on the XML schema documents. The *Bytecode* concept (see Section 9.1.2) might be applicable to EXI. However it currently works with the binary XML format BiM only. Finally, there is *S4G* representing the presented solution that also encodes the XML schema knowledge with EXI but uses the schema for grammars definitions. Neither the candidate *XSD* nor *S4S* meets the requested requirements as requested in Section 9.1.1. Both demand an XML schema parser library to work. Apart from *S4G*, *Bytecode* is the only candidate that removes any dependency on an XML schema parser library.

Figure 9.8 depicts compression numbers. The test set is sorted according to the plain-text XML schema document size. In the case of *S4G* and *Bytecode* it is not possible to simply dismiss the start-up costs of an empty schema. An empty schema accounts for 378 Bytes in the S4G representation while the original textual representation accounts for 111 Bytes and its EXI encoded S4S counterpart accounts for only 3 Bytes. The need to dismiss built-in datatypes of an XML schema parser has a strong negative effect. This is also strengthened by the candidate *Bytecode* that demands even more start-up costs (1867 Bytes). However, already a very simple XML schema document like *Notebook* illustrates the com-

[10]`http://www.zigbee.org/Standards/ZigBeeSmartEnergy/Version20Documents.aspx`



[Bytes]	XSD	S4S	Bytecode	S4G
Empty	111	3	1 867	378
Notebook	648	101	2 060	501
Datastore	939	205	2 165	591
GAML	13 554	1 787	5 145	2 207
WSDL	18 010	3 255	5 009	2 531
V2G	64 192	13 484	20 523	14 004
SEP2	309 132	105 926	39 750	41 051
MPEG-7	347 625	49 577	77 270	198 987

Figure 9.8: Compression Results for XML Schema Exchange Formats

petitiveness of the *S4G* candidate in regard to plain-text XML schema documents by reducing the amount of data that needs to be exchanged.

In general, it can be said that for small schema documents the overhead for the built-in XML schema types is relevant. The more complex and larger the XML schema documents become, the more effective the *schema for grammars* (S4G) solution tends to be.

Looking into the measurements more closely shows why the SEP2 schema is best suited to S4G and stresses that the XML schema uses restrictions that do not affect EXI grammars. Those restrictions are exchanged in the XSD and S4S format but do not change in any way how EXI grammars look like. One example are facets such as `<xs:maxInclusive value="281474976710655"/>`. These and

other similar facets do not affect grammars and hence do not need to be transferred. In fact, this specific facet changes datatype representations and is hence implicit according to the given EXI datatype.

In contrary, we can detect issues with the *MPEG-7* test case. Again, looking into the case more closely highlights the reason. The MPEG-7 schema uses XML schema particles with occurrences such as `maxOccurs="200"` and `maxOccurs="127"`. In Section 9.2.2 we described the correlation of XML schema particle properties and EXI grammars. Larger numbers for XML schema particle occurrences inflate the number of EXI grammar states and productions. Hence, from the EXI perspective, setting `maxOccurs` to `unbounded` is much better suited and can be seen as a schema design guideline for EXI.

Figure 9.9 depicts compression numbers, where an additional compression step (DEFLATE [23]) is affordable. The overall compressed size is much smaller and hence also the amount of data that needs to be exchanged is much smaller. The difference between the formats mostly disappears in regard to size but the trend remains the same as elaborated in Figure 9.8.

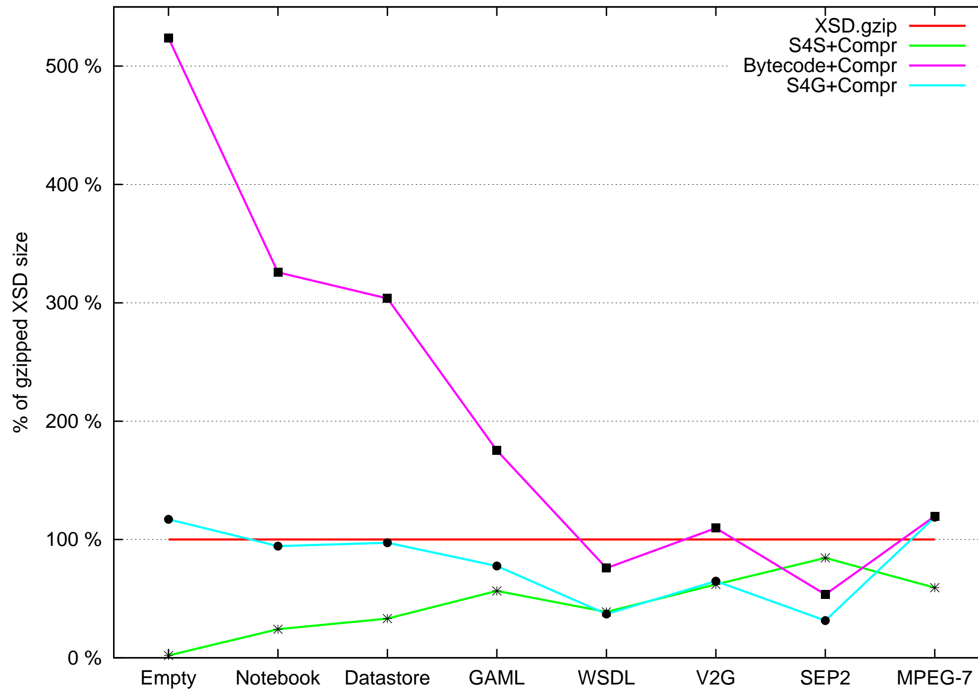
Comparing all compression measurements, the schema for schemas (S4S) solution shows the best compression performance, except for the test case SEP2 and WSDL, where the proposed *S4G* solution is more compact and outperforms all other formats. This means from the point of view of having the most efficient representation on the wire *S4S* is a feasible choice. However, as stressed already, it does not provide a solution for the demanded requirements given that it fully depends on an XML schema parser. That said, the proposed schema for grammars (S4G) solution offers the overall best approach fulfilling all demanded requirements stated in Section 9.1.1.

9.3.4 Parsing Time Measurements

This section discusses the expected time differences between the two ways of building EXI grammars. Figure 9.10 compares the speed of an XML schema library (in our case Xerces-J) to first load an XML schema document and then to build EXI grammars compared to S4G which directly reads the EXI stream according to the schema for grammars definitions to build EXI grammars.

The speed benefit one gets from XSD to S4G highly depends on how much time is spent actually loading the files compared to how much time is spent building EXI grammars.

Figure 9.10 depicts processing performance numbers and highlights that even in the worst case where most of the time is actually spent creating EXI grammars, the time improvement of S4G compared to the baseline XSD is 23%. This means that an EXI processor spends about a third less time waiting for EXI grammars to be ready. However, the MPEG-7 performance highlights once again that given to the particle occurrences, building EXI grammars dominates. The large SEP2



[Bytes]	XSD.gzip	S4S+Compr	Bytecode+Compr	S4G+Compr
Empty	252	5	1 320	295
Notebook	443	107	1 443	418
Datastore	488	162	1 482	475
GAML	1 860	1 050	3 260	1 444
WSDL	4 121	1 597	3 122	1 525
V2G	9 880	6 131	10 824	6 405
SEP2	41 180	34 780	22 053	12 952
MPEG-7	35 417	21 001	42 395	42 082

Figure 9.9: Compression Results for XML Schema Exchange Formats with DEFLATE

test case falls in the same category that most of the time is actually spent creating internal structures and respectively EXI grammars. Hence the time is not the time one format takes to be parsed but mostly the time that is necessary to create the in-memory representations of EXI grammars.

The best performance boost is getting the V2G test case (about 21 times faster) that is due to the fact that the test is composed of several distinct files that XSD or respectively the XML schema parser needs to put together while the S4G solution just deals with one single stream.

For small documents (e.g., *Empty*, *Notebook*, *Datastore*, *GAML* and *WSDL*) the benefit is in the range of two to five times faster for S4G.

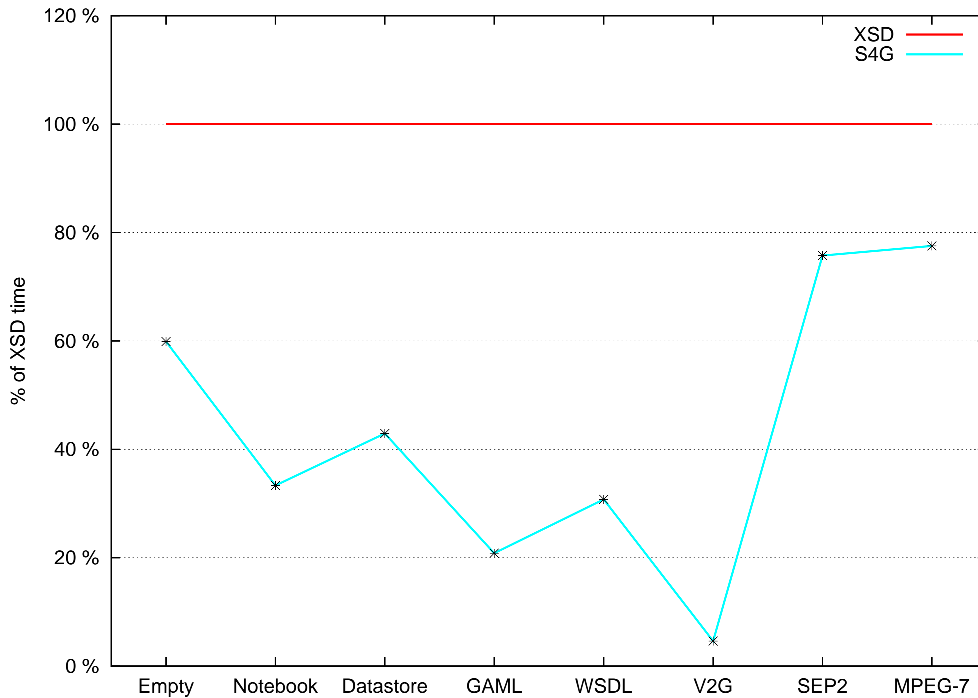


Figure 9.10: EXI Grammars Parsing Time

9.3.5 Summary

Concluding we can say that the proposed *schema for grammars* (S4G) solution offers what is requested. First of all the code footprint of the deployed system can be reduced by the XML schema parser. In our case, when using Java with Xerces-J and EXIficient, this yields a reduction of 1588 kBytes. Further the performance in regard to processing time is much better when compared to parsing an XML schema. An XML schema parser usually first builds an internal model (e.g., XSModel in Xerces-J) which in turn is used to build EXI grammars. Moreover, the S4G solution easily integrates into existing EXI processors and makes them independent of any XML schema parser library. That said, the schema for grammars definition is language independent and is especially designed for fast parsing while also taking into account runtime memory constraints.

Chapter 10

Demand-tailored EXI Processor

The previous Chapter 9 illustrates a very flexible and powerful way to exchange XML schema knowledge required to process schema-informed EXI streams. The presented *schema for grammars* (S4G) approach offers a way to inform an EXI processor with any kind of schema knowledge. In many cases tough, EXI processors do not demand being capable to process any kind of data. This is especially the case for highly restricted device classes. The messages, or respectively the EXI streams that are exchanged, are usually limited to a few different sets of messages. Moreover, only processing support for datatypes that are actually required for those kinds of messages shall be deployed on restricted devices. This allows reducing code footprint and facilitates optimizations that may lead to increased performance. Moreover, lowering the required processing power also reduces costs. Microcontrollers may be less powerful and the memory requirements may be lowered also. These conditions encourage building a *demand-tailored* EXI processor that fits the demanded needs. That said, no more features than actually necessary should be deployed.

The following sections first sketch how a demand-tailored EXI processor may look like. Doing so demands once again analyzing EXI grammars given that EXI grammars build the basis of any EXI processor. Based on the outcome of this research, relations between EXI grammars and EXI events are established that hereinafter lead to how to build a demand-tailored EXI processor. In contrary to what we have seen so far, source code will establish a dedicated EXI processor for a given set of XML schema documents or respectively for a given set of EXI grammars. Until now EXI processors have been examined that are guided by EXI grammars to support schema-informed EXI grammars (e.g., the EXI processor EXIficient). Finally, algorithms are shown that, based on a given set of XML schema documents, build an EXI processor in an automatable way.

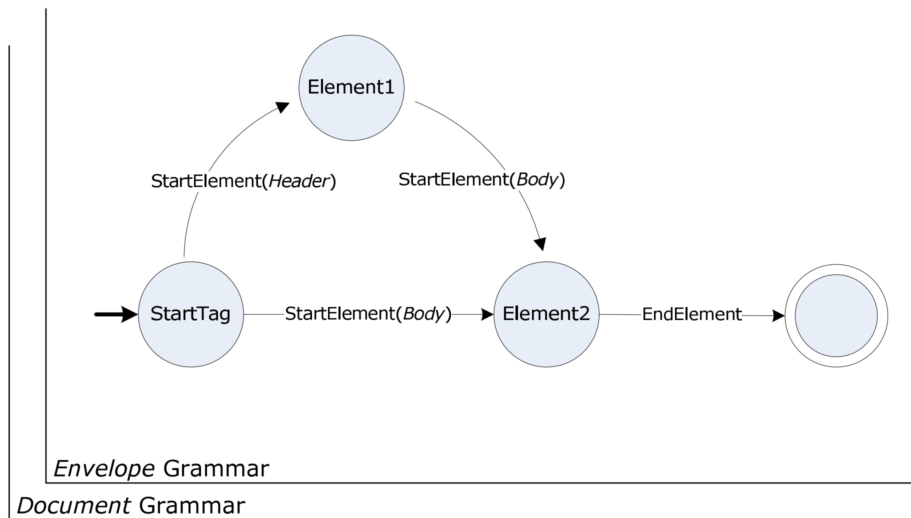


Figure 10.1: EXI Grammar for SOAP element `Envelope` content model

10.1 EXI Grammar Transformations

The EXI format is grammar-driven meaning that processing is based on regular grammars derived from XML schema constraints. The EXI specification [73] defines a predefined process how schema information is to be transformed to EXI grammars.

Figure 10.1 shows exemplarily the grammar of the XML element content model Envelope and respectively the EXI grammar for *Envelope*. The SOAP[1] `Envelope` element is the root element of a SOAP message and allows a nested element `Header` and requires an element `Body`. Given that XML is not a regular language, a single EXI grammar cannot be used to represent an entire XML document. Instead, an EXI processor uses a stack of grammars, one for each element content model. The underlying *Document* grammar stack item in turn describes root elements that may occur in an EXI document.

In general, StartElement (SE) events specify the start of a new XML element while also pushing a new grammar onto the grammar stack. EndElement (EE) events on the other hand pop the current grammar from the stack again. Essentially, EXI offers corresponding events for all XML Information Set items such as characters and attributes.

Chapter 9 illustrated various ways to make a generic EXI processor aware of a set of EXI grammars such as providing grammar knowledge on the basis of XML schema documents or pre-parsed grammar files. Nevertheless, none of the mentioned approaches is suitable for *highly* restricted devices. In addition, the requirements of providing all possible functionalities, datatypes and EXI options

[1] W3C: SOAP Specifications, `http://www.w3.org/TR/soap/`

in each use case (even if not required) is not feasible in microcontroller domains. Hence subsequently a solution, focusing on minimal code footprint and complexity, is presented where the source code implicitly contains all required grammar information without any external dependencies.

10.1.1 Properties of EXI Grammars

EXI grammars correspond to deterministic finite automata (DFA) where events describe the transition from one grammar state to the next state. A transition is to be considered implicit when there is only one possible next state (e.g., Element1 ⇒ Element2 in Figure 10.1). For states where multiple transitions are possible (e.g., *StartTag* provides transitions to *Element1* and *Element2*), the EXI stream indicates with a so called *event code* which path in the automaton has been chosen. This event code is represented by an n-bit unsigned integer ($n = \lceil log_2 m \rceil$, where m is the number of events).

The number of possible events is dictated by the available transitions on the one hand and by the used EXI options on the other hand. In general, the EXI default mode always accounts for one additional *escape* event that is meant to indicate deviations. Such deviations from an XML schema document are accepted and encoded using more generic events. Hence, a deviant XML information, or in other words information not matching the underlying XML schema, can still be represented as an EXI stream, although less efficient. The EXI format also has a special mode called STRICT that implies strict interpretation of grammars (or respectively XML schema) and thus only permits items and datatypes declared in this context. This dedicated mode is used to achieve best compression numbers and decreased processing time by removing deviant information (i.e., *escape* events) and usually unnecessary XML items such as comments. The EXI STRICT mode is also beneficial in regard to processing time due to code simplicity. That said, strict interpretation prunes additional transitions that have been added to deal with deviant data.

It is very likely that most microcontrollers and other highly restricted device classes use strict interpretation of the developed exchange protocol. Hence we subsequently focus on the EXI option STRICT. Nevertheless, all presented mechanisms and algorithms are also applicable to the default EXI mode.

10.1.2 Mapping EXI Grammars to Finite State Machines

A finite state machine (FSM) is a model of behavior composed of a finite number of states, transitions between those states, and actions. It is similar to a *flow graph* where we can inspect the way in which the logic runs when certain conditions are met.

When mapping EXI grammars to the concept of a finite state machine, EXI grammar states represent the FSM states. A transition from one EXI grammar state to the other is indicated by a so called EXI event. EXI has a restricted set of EXI events such as StartDocument (SF), StartElement (SE), Attribute (AT), Characters (CH) EndElement (EE), and EndDocument (ED). The complete list of available EXI events can be retrieved from the official specification website[2].

A so called *event code* triggers which transition has to be taken if more than one transition is available. Hence an event code defines the transition condition that is met. In the case of a single transition only, the event code is implicit and no bit is written to (or read from) an EXI stream.

Despite from moving from one state to the other, most EXI events have an action associated to it (e.g., an AT event stands for an attribute qualified name and the attribute value). Additionally SE and EE events push and pop the current EXI grammar onto or from a stack of grammars. The current grammar is always the peak grammar of the grammar stack.

10.1.3 Mapping EXI Grammars to Source Code

As shortly alluded a generic EXI processor is hardly deployable on microcontroller-based platforms. The code footprint is likely too large and many EXI options account for additional processing and additional code footprint. Recalling the requirements of an EXI processor, that is meant to be successfully deployed on restricted devices, we have to fulfil additional requirements.

- Minimal code footprint.
- Minimal runtime requirements in regard to processing power and memory.
- Support for one (or a restricted) set of XML schema documents.
- Generally, support for one set of EXI options.

These requirements lead to a new way of realizing an EXI processor, different to the ones that were discussed so far and targeting different device capabilities. First of all EXI grammars are not built at runtime or shared in a pre-parsed form. The EXI grammars need to be *built-in* instead, to achieve minimal code footprint. This implies that such an EXI processor is not flexible in supporting various exchange messages but mostly supports instances conforming a given set of XML schema documents (or EXI grammars) only.

Suppose a restricted device such as an Internet-connected watch that is meant to support the running notebook example showing notes on the screen that are pushed to this device. That said, the set of EXI grammars that need to be supported are restricted. Figure 10.2 depicts once again the available set of EXI

[2] Available EXI events, `http://www.w3.org/TR/exi/#eventTypes`

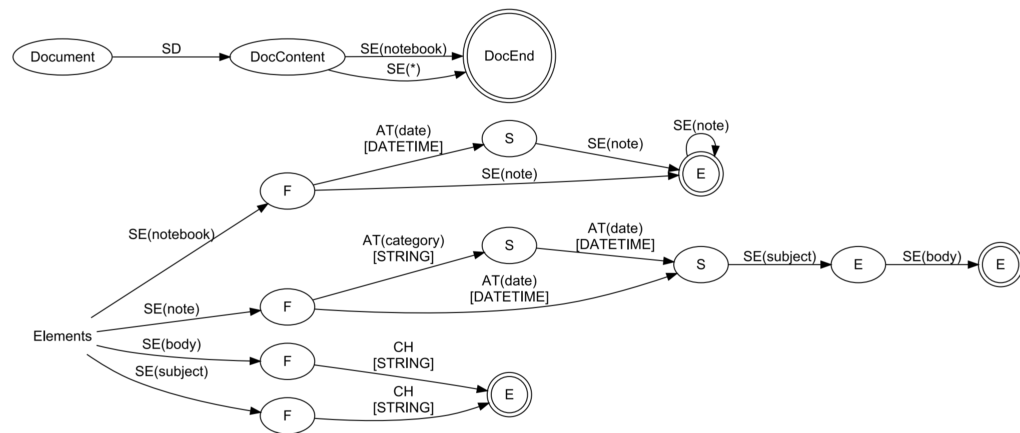


Figure 10.2: Schema-informed EXI grammars for notebook.xsd

grammars. First, we have *Document*, *DocContent* amd *DocEnd* grammars. Further, for each XML element (i.e., `notebook`, `note`, `body`, and `subject`) we have another set of grammars, starting with first start tag grammars denoted by `"F"`, followed by start tag grammars denoted by `"S"`, and element content grammars denoted by `"E"`. To put it briefly, each grammar type relates to different states while processing an XML element. For example in the very beginning of an XML element, the grammar state or respectively the FSM state accounts for attributes (see `"F"` and `"S"`) while after processing characters, attributes are not allowed anymore (see `"E"`). Section 9.2.2 elaborates the differences between the different kinds of grammar types in more details.

The grammars in this example are very simple and have in many states just one transition and at most two possible choices (e.g., *DocContent* grammar).

A developer that is asked to develop a piece of code for parsing (i.e., decoding) EXI streams following the notebook example may likely come up with source code that is similar to what is sketched in Listing 10.1. The method `decodeNBitUnsignedInteger( nbits )` reads `nbits` bits from a given EXI stream. Hence in the listing, it is used to report EXI event codes. The `switch` or `if` statements allow moving from one grammar state to another grammar state. Further we see auxiliary methods for decoding strings or date values (see `decodeString(...)` and `decodeDate(...)`).

Listing 10.1: Restricted Pseudo EXI Source Code Decoder for notebook.xsd

```
/* Start with Document, SD transition implicit to DocContent */
if ( decodeNBitUnsignedInteger( 1 ) == 0 ) {
	/* SE(notebook) */
	switch ( decodeNBitUnsignedInteger( 1 ) ) {
	case 0:
		/* AT(date) */
		decodeDate();
		/* no break */
	case 1:
		/* SE(note) */
		switch ( decodeNBitUnsignedInteger( 1 ) ) {
		case 0:
			/* AT(category) */
			decodeString();
			/* no break */
		case 1:
			/* AT(date) */
			decodeDate();
			/* SE(subject) */
			decodeString();
			/* SE(body) */
			decodeString();
			if( decodeNBitUnsignedInteger( 1 ) == 0 ) {
				/* error: only one note element supported */
			}
		}
		break;
	}
}
```

Consequently, Listing 10.1 provides a very rudimentary sample code that represent an EXI decoder with certain restrictions. That said, the pseudo code supports only EXI streams that are valid to the given notebook.xsd XML schema and are encoded with the default EXI options except that the EXI STRICT mode is established (i.e., no deviations). Further the element `notebook` may contain exactly one `node` element only.

The default EXI stream is bit-packed. Bit-packed means that one bit may indicate one or the other grammar state transition. Moreover, having just one possible transition (e.g., *Document* grammar) implicitly implies this single transition. This is why the pseudo source code directly moves on to the *DocContent* grammar in the very beginning. The *DocContent* grammar has two transitions and hence requires to read one bit ($\lceil log_2 2 \rceil = 1$, see `decodeNBitUnsignedInteger` on line 2). A returned event-code equal to `0` (zero) dictates that the first transition SE(notebook) is taken while an event-code equal `1` (one) would inform about an unexpected element SE(*). In a similar fashion the SE(notebook) grammar is decoded. The next bit informs whether an attribute AT(date) is available that would also imply decoding the date value.

The presented code is limited to a certain set of messages but it also gives a very good idea how simple an EXI processor (in this case the decoder) can be built. However, the presented approach also shows some problems that may need to be solved before being able to successfully create source code for more complex

EXI grammars. One main issue is the capability of moving from one grammar state back to a previous one which was the reason that only one `note` element is supported. This situation accounts for either duplicate code or a mean to establish some kind of jump labels (see Listing 10.1 where in line 24 the program needs to jump back to line 4). The first solution of duplicating code is only sensible for very simple EXI grammars and practically impossible for occurrences that are unbounded. Hence we focus on efficiently creating jump labels. Moreover, the ideal case is that an EXI processor can be created with a given automatable generation process following pre-defined steps and algorithms.

10.2 Automatic EXI Processor Generation

The previous section sketched the mechanism that can be used to produce a program or library that decodes XML information represented as EXI stream. Vice versa, XML information items can be marshalled to an EXI stream. For doing so, three steps are involved. To support automatic generation of such EXI processors the following steps need to be taken into account.

1. **Analyze XML schema information:** The widely adopted XML schema processor Xerces-J is used to analyze XML schema documents. As a result, all possible XML elements, attributes, and constraints in this specific schema context are obtained.

2. **Build EXI grammars according to schema constructs and datatypes:** The EXI specification [73] accurately describes the process of creating EXI grammars from schema constructs. The efforts in the open source EXI implementation EXIficient already prepared the technical ground for this process.

3. **Create source code according to EXI grammars:** This is the novel step that is subsequently explained in more detail.

10.2.1 Automatic Source Code Generation according to EXI Grammars

The presented concepts do not focus on code generation for a specific programming language. Instead, the use cases in mind are to support various programming languages such as ANSI-C, C++, and Java. Some of our test beds run on Java Micro Edition. Others base on Contiki, an open source operating system for networked, memory-constrained systems with a particular focus on low-power wireless devices. Yet others use nesC[3].

[3] nesC (network embedded systems C) is a component-based, event-driven programming language used to build applications for the TinyOS platform.

The extensive application use within various programming languages demands that the presented mechanisms are flexible enough to produce source code in any programming language. That said, the concepts ought to be generally applicable.

EXI grammars consist of a set of grammar states. Production rules define relations among each state. A very effective way to identify grammar states is to uniquely assign labels. For example the grammar state *Document* is labeled with 0, *DocContent* with 1, *DocEnd* with 2, and the initial state `F` for the notebook element with 3 et cetera. This process continues for all EXI grammar states.

Figure 10.3: Schema-informed EXI grammars for notebook.xsd with grammar IDs

Figure 10.3 depicts exemplarily this process of labelling grammar states with IDs (grammar IDs are depicted within angle brackets, e.g., `[1]`). This unique IDs can be used in switch case statements as jump labels. We will shortly see an example how this may look like.

However, first we also need to solve the second issue of the previously introduced initial sample, namely structuring the source code into smaller and manageable portions. This highly relates to the application programming interface.

10.2.2 Application Programming Interface

Traditionally, XML application programming interfaces (APIs) are either *DOM-based* [56]—the entire document is read into memory as a tree structure—or *event-based push-APIs* (e.g., SAX [12])—the application registers to receive events as entities are encountered within the document. Both have advantages; the former allows for random access to the document, the latter comes with a smaller memory footprint and is typically much faster.

Pull APIs (e.g., StAX [31]) were designed as a median between these two opposites. The entry point is a cursor that represents a point within the document. The application moves the cursor forward—*pulling* the information from the parser as it needs. Many microcontroller based use cases demand pull APIs given that the memory is not sufficient to store all information. Moreover, the limited application requires to be able to decide when being capable to receive new data. Hence, a pull API is able to move on the cursor in the stream and reports the according XML information such as elements, attribute, and character data.

Listing 10.2: Typed API for EXI Decoder (Excerpt)

```
1
2  /**
3   * \brief Inspects EXI stream and decodes next EXI event.
4   */
5  int exiDecodeNextEvent(bitstream_t* stream,
6                 exi_state_t* state, exi_event_t* nextEvent);
7
8
9  /**
10  * \brief Decodes StartDocument (SD) event
11  */
12 int exiDecodeStartDocument(bitstream_t* stream,
13                exi_state_t* state);
14
15 /**
16  * \brief Decodes EndDocument (ED) event
17  */
18 int exiDecodeEndDocument(bitstream_t* stream,
19                exi_state_t* state);
20
21 /**
22  * \brief Decodes StartElement (SE) event
23  */
24 int exiDecodeStartElement(bitstream_t* stream,
25                exi_state_t* state, uint16_t* qnameID);
26
27
28 /**
29  * \brief Decodes EndElement (EE) event
30  */
31 int exiDecodeEndElement(bitstream_t* stream,
32                exi_state_t* state, uint16_t* qnameID);
33
34
35 /**
36  * \brief Decodes Characters (CH) event.
37  */
38 int exiDecodeCharacters(bitstream_t* stream,
39                exi_state_t* state, exi_value_t* val);
40
41
42 /**
43  * \brief Decodes Attribute (AT) event.
44  */
45 int exiDecodeAttribute(bitstream_t* stream,
46                exi_state_t* state, uint16_t* qnameID, exi_value_t* val);
```

Listing 10.2 introduces the pull API in ANSI-C for parsing an EXI stream that will be used in the following sections. Apart from setting up the EXI stream itself and initializing the decoder the given interface provides a method for inspecting the stream and reporting the next EXI event (see `exiDecodeNextEvent(...)`). Based on the according EXI event that can be StartDocument (SD), EndDocument (ED), StartElement (SE), EndElement(EE), Characters (CH), or Attribute (AT), the matching decode method needs to be called. For example, an EXI event `ATTRIBUTE` implies calling `exiDecodeAttribute(...)` that in turn reports the attribute qualified name and the attribute value.

Further, the general assumption of this error reporting API is that each function call reports as return value success or failure. Any return value other than zero (0) is interpreted as failure. The function parameter `stream` typed as `bitstream_t` is the EXI input stream and the function parameter `state` typed as `exi_state_t` implies the EXI processor state.

Subsequently, a procedure for each relevant API function is given first in the form of an algorithm and second as ANSI-C source code. Note that the procedure is flexible enough to produce code in any programming language. However, in our case ANSI-C has been chosen.

10.2.3 EXI Decoder Generation

For simplicity, the main focus is to show how the algorithms for creating an EXI parser works. The EXI parser (sometimes also called EXI decoder) process can be applied in a very similar fashion to the encoder side as well.

EXI Decoder - Next Event

The algorithm that is used is rather simple and straight forward. We will also see some optimization potential later. The concept is based on a list of EXI grammars where the grammarID is the unique position of each grammar in the grammar list $(0...numberofgrammars - 1)$.

The source code creation process in Algorithm 1 walks over the grammar list and `prints` output in a programming-language-dependent way. For each grammarID a case statement is created so that an EXI processor, depending on the current grammarID, can directly jump to the relevant portion of the code. The **if** block starting on line 12 deals with grammars where a single transition is available. Hence the transition is implicit and there is no need to decode any event-code from the EXI stream. The **else** part starting on line 15 handles cases where we have multiple possible transitions. According to the number of transitions (see $numberOfTransitions$), the logarithm to base 2 is built, which defines the number of bits that need to be read to decode the right EXI event-code. Also, the number of transitions for each grammar state form another nested switch do

Algorithm 1 EXI Decoder Generation - Next Event algorithm

```
 1: procedure NextEvent(grammarList)                       ▷ Report the next EXI event
 2:     print 'int exiDecodeNextEvent(...)  {'             ▷ Function Definition
 3:     print 'int16_t grammarID = state->grammarStack[state->stackIndex];'
 4:     print 'state->eventCode = 0;'
 5:     print 'errn = 0;'
 6:     print 'switch (grammarID) {'
 7:     for i = 0 to sizeof(grammarList) − 1 do
 8:         print 'case ' + i + ':'
 9:         currGrammar ← grammarList[i]
10:         transitionList ← getTransitions(currGrammar)
11:         numberOfTransitions ← sizeof(transitionList)
12:         if numberOfTransitions = 1 then
13:             event ← getEvent(transitionList[0])
14:             print '*nextEvent = ' + event + ';'
15:         else
16:             nbits ← log 2 numberOfTransitions
17:             print 'errn = decodeNBitUnsignedInt..' + nbits + ', ..);'
18:             print 'switch ( state->eventCode ) {'
19:             for k = 0 to numberOfTransitions − 1 do
20:                 event ← getEvent(transitionList[k])
21:                 print 'case ' + k + ':'
22:                 print '*nextEvent = ' + event + ';'
23:                 print 'break;'                         ▷ Inner Break
24:             end for
25:             print '}'
26:         end if
27:         print 'break;'                                 ▷ Outer Break
28:     end for
29:     print '}'
30:     print 'return (errn);'
31:     print '}'                                          ▷ End of Function
32: end procedure
```

deal with all available event transitions. Note that the logarithm computation is done at compile time and not at runtime.

Listing 10.3 presents the output of the source code generation algorithm for the ANSI-C programming language. The function `exiDecodeNextEvent(...)` is called whenever an application wants to inspect the EXI stream to retrieve the next EXI event. According to the current grammarID, the outer switch jumps to the relevant case statement. For example grammarID set to 0 (zero) is the start of the stream that deals with the *Document* grammar. The *Document* grammar has exactly one possible transition, namely StartDocument (SD) which is implicit. Once an application calls the `exiDecodeStartDocument(...)` func-

tion (not shown here) the current grammarID changes to 1 (one). Hence we deal with the *DocContent* grammar. The *DocContent* grammar disposes of two possible transitions and depending on the decoded event-code the next EXI event is reported. This process is similar for all other grammars (the grammarIDs range from 0 ... 13).

Listing 10.3: EXI Decoder - Next Event

```
1  int exiDecodeNextEvent(bitstream_t* stream, exi_state_t* state,
2                  exi_event_t* nextEvent) {
3          int16_t grammarID = state->grammarStack[state->stackIndex];
4          state->eventCode = 0;
5          errn = 0;
6
7          switch (grammarID) {
8          case 0:
9                  /* Document[START_DOCUMENT] */
10                 *nextEvent = EXI_EVENT_START_DOCUMENT;
11                 break;
12         case 1:
13                 /* DocContent[START_ELEMENT(notebook),
                       START_ELEMENT_GENERIC] */
14                 errn = decodeNBitUnsignedInteger(stream, 1, &state->
                       eventCode);
15                 switch (state->eventCode) {
16                 case 0:
17                         *nextEvent = EXI_EVENT_START_ELEMENT;
18                         break;
19                 case 1:
20                         *nextEvent = EXI_EVENT_START_ELEMENT_GENERIC;
21                         break;
22                 }
23                 break;
24         case 2:
25                 /* DocEnd[END_DOCUMENT] */
26                 *nextEvent = EXI_EVENT_END_DOCUMENT;
27                 break;
28         case 3:
29                 /* FirstStartTag[ATTRIBUTE[DATETIME](date), START_ELEMENT(
                       note)] */
30                 errn = decodeNBitUnsignedInteger(stream, 1, &state->
                       eventCode);
31                 switch (state->eventCode) {
32                 case 0:
33                         *nextEvent = EXI_EVENT_ATTRIBUTE;
34                         break;
35                 case 1:
36                         *nextEvent = EXI_EVENT_START_ELEMENT;
37                         break;
38                 }
39                 break;
40         case 4:
41                 /* StartTag[START_ELEMENT(note)] */
42         case 8:
43                 /* StartTag[START_ELEMENT(subject)] */
44         case 9:
45                 /* Element[START_ELEMENT(body)] */
46                 *nextEvent = EXI_EVENT_START_ELEMENT;
47                 break;
48         case 5:
49                 /* Element[START_ELEMENT(note), END_ELEMENT] */
```

```
50                  errn = decodeNBitUnsignedInteger(stream, 1, &state->
                        eventCode);
51                  switch (state->eventCode) {
52                  case 0:
53                          *nextEvent = EXI_EVENT_START_ELEMENT;
54                          break;
55                  case 1:
56                          *nextEvent = EXI_EVENT_END_ELEMENT;
57                          break;
58                  }
59                  break;
60          case 6:
61                  /* FirstStartTag[ATTRIBUTE[STRING](category), ATTRIBUTE[
                        DATETIME](date)] */
62                  errn = decodeNBitUnsignedInteger(stream, 1, &state->
                        eventCode);
63                  switch (state->eventCode) {
64                  case 0:
65                  case 1:
66                          *nextEvent = EXI_EVENT_ATTRIBUTE;
67                          break;
68                  }
69                  break;
70          case 7:
71                  /* StartTag[ATTRIBUTE[DATETIME](date)] */
72                  *nextEvent = EXI_EVENT_ATTRIBUTE;
73                  break;
74          case 10:
75                  /* Element[END_ELEMENT] */
76          case 12:
77                  /* Element[END_ELEMENT] */
78                  *nextEvent = EXI_EVENT_END_ELEMENT;
79                  break;
80          case 11:
81                  /* First(xsi:type)StartTag[CHARACTERS[STRING]] */
82          case 13:
83                  /* First(xsi:type)StartTag[CHARACTERS[STRING]] */
84                  errn = decodeNBitUnsignedInteger(stream, 1, &state->
                        eventCode);
85                  switch (state->eventCode) {
86                  case 0:
87                          *nextEvent = EXI_EVENT_CHARACTERS;
88                          break;
89                  case 1:
90                          /* 2nd level events */
91                          *nextEvent = EXI_EVENT_ATTRIBUTE_XSI_TYPE;
92                          break;
93                  }
94                  break;
95
96          }
97
98          return (errn);
99  }
```

Line 40 and the following lines of Listing 10.3 show another interesting aspect of optimizations. The case statements of #4, #8, and #9 share the same code portion. Hence the code can be grouped together, which reduces code footprint for larger grammar sets. The same applies to the case statements #10 and #12 as well as the case statements #11 and #13.
The algorithm that is used to detect shared code portion is very simple but effective. The code portions are generated and indexed . In the case of outputting

the same functionality (e.g., twice an EndElement (EE) event) exactly the same code portion is generated (character by character). Hence source code portions can be combined with multiple cases if the source code portions compare equal to previous appearances.

After initializing an EXI stream, the presented decode `exiDecodeNextEvent(...)` method is used to move forward the pull-API cursor to the next event. In the following subsections, an analog algorithm for other prominent EXI events (start elements, characters, and attributes values) is given. According to each event the dedicated method needs to be called (i.e., for `EXI_EVENT_START_ELEMENT` `exiDecodeStartElement(...)` needs to be called).

EXI Decoder - Start Element

The next algorithm deals with start elements and analyzes the parsing side, whereas the serializer can be constructed in a similar fashion.

Algorithm 2 for decoding start elements follows a similar approach to Algorithm 1. It walks over all EXI grammars. However, it only generates code for EXI events that relate to start elements. According to the given known start element event and the correlation to the qualified name the URI and the local-name of the specific event is given [4]. Further, for each start element a new stack item is pushed onto the grammar stack. The top of the stack depicts the current grammar and its ID. EE (End Element) events pop the stack by one grammar again.

The generated code output in Listing 10.4 for the notebook.xsd schema creates decoding methods for each known element, namely `notebook`, `note`, `body`, and `subject`. Case statements are created for start elements only while other events in this specific context are not of interest and therefore not taken into account.

[4] XML schema wildcards lead to grammar productions with absent URI and/or local-name (see `http://www.w3.org/TR/exi/#wildcardTerms`) . EXI grammars, and respectively the algorithm, provide dedicated productions which are not addressed in this section. For such start element events, the according qualified *local-name* and/or *URI* need to be decoded also. Other than that, the general processing remains the same.

Algorithm 2 EXI Decoder Generation - Start Element (SE) algorithm

```
 1: procedure StartElement(grammarList)                ▷ Decode StartElement (SE)
 2:     print 'int exiDecodeStartElement(...) {'        ▷ Function Definition
 3:     print 'int16_t grammarID = state->grammarStack[state->stackIndex];'
 4:     print 'errn = 0;'
 5:     print 'switch (grammarID) {'
 6:     for i = 0 to sizeof(grammarList) − 1 do
 7:         currGrammar ← grammarList[i]
 8:         transitionList ← getTransitions(currGrammar)
 9:         numberOfTransitions ← sizeof(transitionList)
10:         print 'switch(state->eventCode) {'
11:         for k = 0 to numberOfTransitions − 1 do
12:             event ← getEvent(transitionList[k])
13:             if event is START_ELEMENT then
14:                 print 'case ' + k + ':'
15:                 print 'errn = _exiDecodeStartElement(...);'
16:                 print 'break;'                          ▷ Inner Break
17:             end if
18:         end for
19:         print '}'
20:         print 'break;'                                  ▷ Outer Break
21:     end for
22:     print '}'
23:     print 'return (errn);'
24:     print '}'                                           ▷ End of Function
25: end procedure
```

Listing 10.4: EXI Decoder - Start Element

```
1  int exiDecodeStartElement(bitstream_t* stream, exi_state_t* state,
2                uint16_t* qnameID) {
3          int16_t grammarID = state->grammarStack[state->stackIndex];
4          errn = 0;
5
6          switch (grammarID) {
7          case 1:
8                  /* DocContent[START_ELEMENT(notebook),
                       START_ELEMENT_GENERIC] */
9                  switch(state->eventCode) {
10                 case 0:
11                         errn = _exiDecodeStartElement(state, *qnameID = 5,
                               2, 3);
12                         break;
13                 }
14                 break;
15         case 3:
16                 /* FirstStartTag[ATTRIBUTE[DATETIME](date), START_ELEMENT(
                       note)] */
17                 switch(state->eventCode) {
18                 case 1:
19                         errn = _exiDecodeStartElement(state, *qnameID = 4,
                               5, 6);
20                         break;
```

```
21                  }
22                  break;
23          case 4:
24                  /* StartTag[START_ELEMENT(note)] */
25                  switch(state->eventCode) {
26                  case 0:
27                          errn = _exiDecodeStartElement(state, *qnameID = 4,
                                5, 6);
28                          break;
29                  }
30                  break;
31          case 5:
32                  /* Element[START_ELEMENT(note), END_ELEMENT] */
33                  switch(state->eventCode) {
34                  case 0:
35                          errn = _exiDecodeStartElement(state, *qnameID = 4,
                                5, 6);
36                          break;
37                  }
38                  break;
39          case 8:
40                  /* StartTag[START_ELEMENT(subject)] */
41                  switch(state->eventCode) {
42                  case 0:
43                          errn = _exiDecodeStartElement(state, *qnameID = 6,
                                9, 13);
44                          break;
45                  }
46                  break;
47          case 9:
48                  /* Element[START_ELEMENT(body)] */
49                  switch(state->eventCode) {
50                  case 0:
51                          errn = _exiDecodeStartElement(state, *qnameID = 1,
                                10, 11);
52                          break;
53                  }
54                  break;
55          }
56
57          return (errn);
58  }
```

EXI Decoder - Characters

The decoder call for characters in Listing 10.5 is comparable to what we haven seen for start elements. The difference is that instead of providing decoding functionalities for elements, decoding functionalities for characters are given. Figure 10.3 depicts that for our running sample schema we only account for two grammar states where we expect EXI characters. The elements subject and body have an EXI event Characters (CH) and both are typed as string. Again, this makes it possible to apply the optimization of collapsing both grammar IDs (#11 and #13) and respectively both code portions to one.

Listing 10.5: EXI Decoder - Characters

```
1  int exiDecodeCharacters(bitstream_t* stream, exi_state_t* state,
2                  exi_value_t* val) {
3          int16_t moveOnID = 0;
```

```
4           errn = EXI_ERROR_UNEXPECTED_CHARACTERS;
5   
6           switch (state->grammarStack[state->stackIndex]) {
7           case 11:
8           case 13:
9                   /* STRING */
10                  val->type = EXI_DATATYPE_STRING;
11                  errn = decodeStringValue(stream, state, state->
                        elementStack[state->stackIndex], &val->str);
12                  moveOnID = 12; /* move on ID */
13                  break;
14          }
15          if (errn == 0) {
16                  /* move on */
17                  state->grammarStack[state->stackIndex] = moveOnID;
18          }
19  
20          return (errn);
21  }
```

EXI Decoder - Attribute

Attribute decoding works again very similar. For each known attribute, an according decoding function is established with the appropriate datatype. The known `qnameID` or respectively the correlated qualified name URI and a local name are also set.

Listing 10.6: EXI Decoder - Attribute

```
1   int exiDecodeAttribute(bitstream_t* stream, exi_state_t* state,
2                   uint16_t* qnameID, exi_value_t* val) {
3           int16_t moveOnID = 0;
4           int16_t currentID = state->grammarStack[state->stackIndex];
5           errn = EXI_ERROR_UNEXPECTED_ATTRIBUTE;
6   
7           switch (currentID) {
8           case 3:
9                   /* FirstStartTag[ATTRIBUTE[DATETIME](date), START_ELEMENT(
                        note)] */
10                  switch(state->eventCode) {
11                  case 0:
12                          *qnameID = 3;
13                          val->type = EXI_DATATYPE_DATETIME;
14                          errn = decodeDateTime(stream, EXI_DATETIME_DATE, &
                                val->datetime);
15                          moveOnID = 4; /* move on ID */
16                          break;
17                  }
18                  break;
19          case 6:
20                  /* FirstStartTag[ATTRIBUTE[STRING](category), ATTRIBUTE[
                        DATETIME](date)] */
21                  switch(state->eventCode) {
22                  case 0:
23                          *qnameID = 2;
24                          val->type = EXI_DATATYPE_STRING;
25                          errn = decodeStringValue(stream, state, *qnameID, &
                                val->str);
26                          moveOnID = 7; /* move on ID */
27                          break;
28                  case 1:
```

```
29                          *qnameID = 3;
30                          val->type = EXI_DATATYPE_DATETIME;
31                          errn = decodeDateTime(stream, EXI_DATETIME_DATE, &
                                val->datetime);
32                          moveOnID = 8; /* move on ID */
33                          break;
34                  }
35                  break;
36          case 7:
37                  /* StartTag[ATTRIBUTE[DATETIME](date)] */
38                  switch(state->eventCode) {
39                  case 0:
40                          *qnameID = 3;
41                          val->type = EXI_DATATYPE_DATETIME;
42                          errn = decodeDateTime(stream, EXI_DATETIME_DATE, &
                                val->datetime);
43                          moveOnID = 8; /* move on ID */
44                          break;
45                  }
46                  break;
47          }
48
49          if (errn == 0) {
50                  /* move on */
51                  state->grammarStack[state->stackIndex] = moveOnID;
52          }
53
54          return (errn);
55  }
```

In summary, it can be concluded that the presented decoding methods build the entire EXI decoder for the given notebook.xsd XML schema document. The outcome is minimal in regard to source code and code footprint and is also very efficient in regard to processing.

10.2.4 Automatic Databinding

So far, most application programming interfaces working with XML have been text-based. In the case of XML attributes or characters, values are reported as strings and it was up to the application to convert textual data such as "123" into its typed representation of an integer.

In Section 10.2.2, we introduced our typed API. Moreover, with the schema knowledge we not only provide typed data in the first place but we also allow automatic databinding. Under the term *automatic databinding* a mechanism is understood that allows filling data containers such as C structs or Java/C++ classes automatically. In the context of EXI streams, the data itself is reported type-aware and also reports the surrounding qualified name. Listing 10.7 depicts an example XML fragment where an EXI decoder reports integer and float values.

Listing 10.7: XML fragment for typed values

```
<values>
```

```
        <intValue>123</intValue>
        <floatValue>4.56</floatValue>
</values>
```

Having XML schema knowledge it is common to build data containers for XML documents in any programming language (e.g., C struct in Listing 10.8). This allows applications to access the data in the usual language-dependent form rather than using XML APIs to retrieve the data from the XML itself. That said, the afore mentioned automatic databinding implies filling the data containers so that developers do not need to worry anymore about serializing or de-serializing EXI. The developer needs to deal with programming language containers such as C structs only. The EXI processing is up to the code generation step.

Listing 10.8: Databinding mapping for <values /> XML fragment

```
1 struct values {
2         int intValue;
3         float floatValue;
4 }
```

In collaboration with Sebastian Käbisch et al. [46] a real-world use case with temperature and humidity sensors has been elaborated that serves for XML-based Web service communication on microcontroller-based devices. In this use case we proved the applicability of the proposed automatic databinding also with regard to processing time (i.e., responsiveness).

10.3 Results and Discussion

To confirm the efficiency, in regard to code footprint and processing time, of the presented algorithm for automatically generating EXI processors, a test bed was set up.

10.3.1 Test Candidates

For the measurements, widely-used and well-known XML parsers written in C[5] were taken into account. In the area of EXI processors there is no such variety. However, one other relevant EXI processor has been analyzed that meets the requirements for resource constrained environments.

Expat XML Parser

Expat[6] is an XML parser library written in C. It is a stream-oriented parser in which an application registers handlers for things the parser might find in the

[5] Literature for microcontrollers such as [54] usually recommends using assembly and C.

[6] Expat website: http://expat.sourceforge.net/

XML document (like start tags). It is considered to be a very fast and low-footprint XML parser library for small or embedded applications. The version that has been tested is Expat 2.0.1.

Libxml2

Libxml2[7] is considered to be one of the widely used, mature, and extensive (support for SAX, DOM, and validation) XML parser libraries. It is the standard XML library of GNU and written in the C programming language. Bindings to other programming languages such as C++, C#, and Phyton are available. The version that has been used is libxml2-2.7.8.

Please remind that other widely used XML parsers such as Xerces-C++[8] have not been taken into account given that the source code is written in C++ or other programming languages that do not seem feasible for restricted devices.

EXIP

EXIP[9] is a project started at EISLAB research group in the Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology. It is part of research efforts to bring resource-constrained embedded devices, such as wireless sensor nodes, closer to the enterprise business processes taking place in processing, manufacturing, and communication industries [45].

The open source project is meant to provide a free and fully-featured open source C language implementation of the EXI specification for resource-constrained embedded devices. The version that has been used is 0.4.1.

EXIdizer

EXIdizer is the given name for the presented source code generator. EXIdizer is able to produce a fully featured EXI processor for almost any kind of XML schema[10]. The goal is to generate source code for a given XML schema document only. Hence, according to the given XML schema documents, a demand-tailored EXI processor is built that supports necessary EXI grammars and EXI datatypes (according to the schema constructs) only.

---

[7] Libxml2 website: `http://xmlsoft.org/`

[8] Xerces-C++ website: `http://xerces.apache.org/xerces-c/`

[9] EXIP sourceforge website: `http://exip.sourceforge.net/`

[10] The datatype support of EXIdizer is limited to finite value spaces. For example the XML schema datatype `xs:integer` has an infinite value space {...,-2,-1,0,1,2,...} and is mapped to int64_t.

10.3.2 Code Footprint Numbers

Compiler translate statements into CPU instructions. Declarations of static data are translated into machine-specific data specifications. To create an executable, the linker aggregates the instructions and the data into distinct segments. All instructions go into one segment traditionally called *text*. In contrast to what the name conveys, the segment contains no source code. The data segment is arranged in two sub-segments. One is called *data*, for the initialized static data and literal constants, and the other is called *bss*, for the uninitialized static data.

Table 10.1 lists the associated libraries that are necessary to actually build an XML parser or respectively an EXI parser. As it turns out Libxml2 is very demanding in regard to code footprint, especially compared to the XML candidate Expat, which is about one-eighth in size. The EXI library for EXIP is also rather big while the test candidate EXIdizer does not demand any external library. We will see subsequently the code footprint for a sample application.

Table 10.1: XML/EXI parser library size (in Bytes)

Size	Expat	Libxml2	EXIP	EXIdizer
text	126 882	981 317	316 509	-
data	3 072	4 096	58 692	-
bss	0	0	120	-

An XML parser implementation that uses either Expat or Libxml2 bases on the referenced libraries and the provided interfaces (e.g., SAX). An implementation thereof can then be used for various XML documents. Also, since there is no possibility to integrate XML schema support, the XML parser does not depend on schema document instances. Use cases that demand mapping textual data to typed representations need to build that capability on top of the XML parser.

The candidate EXIP and its EXI parser instance provide two mechanisms to work, namely in a schema-less and in a schema-informed fashion. Similar to the XML parsers, EXIP may work without schema information. Doing so all data are typed as strings. However, in addition to this schema-less mode, it also supports schema-informed processing. When the schema-enabled mode is set the EXIP parser must be informed by XML schema information. The schema information needs to be encoded with EXI since there is no support for neither reading nor writing XML. This approach is similar to what has been presented in the previous Chapter (see Section 9.1.2). Hence, EXIP requires an XML schema processor that is built-in. This might explain the rather big library size listed in Table 10.1. Despite its size, the EXIP project still does not claim to support all XML schema constructs and/or facets. Moreover, converting XML schema documents to EXI needs to be done with external tools that are able to create EXI streams of any XML input document (e.g., EXIficient).

EXIdizer, as depicted in Table 10.1, does not depend on any library. Depending on the used EXI options and settings (e.g., available schema information and EXI coding modes), the according EXI processor (parser and serializer) is generated. For example, if according to the schema documents there is no need to support float values no float datatype support is provided. Doing so provides the smallest possible processor with no external dependencies.

Parser Example Application

To be able to show the actual code footprint and processing numbers for an application and to give a good comparison between the test candidates a very simple use case will be considered. The test scenario (referred to as *TestApp* in Table 10.2 and Table 10.3) is about counting the number of elements, attributes, and character values within an XML document or respectively within an EXI stream.

This very simple scenario is aimed to give an impression about the additional demand in regard to source code for supporting this use-case. More complex use cases such as parsing typed data are not accomplishable by all candidates. For example XML parser candidates are not aware of any schema information. Hence it is not feasible for an XML parser to convert the reported character sequences to typed information such as integer numbers or decimal values. However, the schema-informed EXIP processor and EXIdizer do provide type information already as built-in feature. Hence, integers are reported as integer values and not as *text* as XML parsers would do.

For comparison reasons, the same set of XML schema documents as in the previous Chapter (Section 9.3.2) has been selected. Starting from no-schema information, also the running XML schema notebook.xsd has been chosen. Further, Datastore and GAML schema documents were taken from the W3C working group directory[11]. Moreover, relevant microcontroller-based use cases such as V2G (ISO/IEC 15118 [3]) and SEP2 [7] have been selected.

The subsequently mentioned footprint numbers have been acquired on a Windows PC. Moreover, the test application has been compiled with GCC 4.6.2 with the compile optimization flags `-O3` and `-Os` set.

Table 10.2 shows numbers for the XML parser candidates, namely Expat and Libxml2. The *shared* library size is listed once again beside the test parser application (TestApp). The pure test application sizes between the two XML candidates do not differ much. However, to be able to run the test XML parsing Expat takes about 130 kBytes while Libxml2 accounts for almost 1 MByte.

Table 10.3 shows the numbers for the EXI parser candidates. In a similar fashion, the table lists the size of the library and the test application. For the test candidate EXIP, the test application remains the same for all different input

[11] EXI Measurement Test Framework, `http://www.w3.org/XML/EXI/#TestingFramework`

Table 10.2: XML parser requirements (size in Bytes)

	Expat		**Libxml2**	
Size	Library	TestApp	Library	TestApp
text	126 882	4 092	981 317	4 332
data	3 072	1 148	4 096	1 248
bss	0	128	0	128

documents. The difference for each instance document is the *exified* XML schema document (see row XSD in Table 10.3) that needs to be passed to EXIP to allow schema-informed processing. Hence, doing so allows EXIP to process schema-informed EXI streams. For example, to process an EXI stream that has been generated with the schema information (e.g., notebook.xsd), EXIP accounts for the library, the test application, and the according notebook schema information. The EXIP developers plan to support more possibilities to inform the processor about the XML schema information. Currently EXIP supports only XML schema definitions represented in EXI format (see EXIP User Guide[12]).

The candidate EXIdizer works slightly different. There is no shared library or any other external data the EXI processor depends on. The EXI processor is purely built based upon the schema information. As a special case also no schema information can be passed to the generation tool and it produces a schema-less EXI processor (see column Schema-less in Table 10.3) that can be used for arbitrary schema-less EXI streams. If schema-informed EXI streams are to be processed the according processor is built. As example, to be able to process schema-informed EXI streams that are encoded with notebook.xsd schema information, the processor accounts for only about 25 kBytes. That said, the actual size of the processor depends on the complexity of the given schema. The more EXI grammar states and EXI events need to be processed the larger the EXI processor becomes.

According to the given numbers in Table 10.3, we conclude that the test candidate EXIdizer is in all cases the smallest EXI candidate in regard to code footprint. For less complex schema documents the difference is large and becomes smaller for very complex schema definitions. However, restricted devices do not tend to be used for complex scenarios anyway.

Table 10.4 illustrates the number of EXI grammar states and the number of EXI grammar events. It is easy to grasp that the complexity of the given EXI grammar highly correlates with the code footprint size of the generated EXI processor EXIdizer listed in Table 10.3.

[12]EXIP User Guide: `http://exip.sourceforge.net/exip-user-guide.pdf`

Table 10.3: EXI parser requirements based on the used schema information (size in Bytes)

Size	**EXIP** Library	TestApp	Notebook	Datastore	GAML	V2G	SEP2
text	316 509	6 372	-	-	-	-	-
data	58 692	1 440	-	-	-	-	-
bss	120	132	-	-	-	-	-
XSD			262	396	2 028	15 614	105 219
Size	**EXIdizer** Library	Schema-less	Notebook	Datastore	GAML	V2G	SEP2
text	-	16 180	23 288	25 288	35 720	85 236	327 848
data	-	1 540	1 784	1 808	1 912	3 384	4 376
bss	-	600	784	784	844	1 556	2 080

Table 10.4: Number of EXI grammar states and events

	# of grammars	# of events
Notebook	57	70
Datastore	57	84
GAML	125	238
V2G	440	1 095
SEP2	1 213	5 485

10.3.3 Parser Performance

The parser performance in regard to processing time has been also measured. XML instances matching the given XML schema documents have been selected and the processing time compared to all candidates has been analyzed. The time in Figure 10.4 has been normalized according to the fastest processor. That said, it turns out that EXIdizer's processing time is the lowest in all cases. Moreover, the API reports already typed data while XML parser candidates do report characters that in an actual representation need to be further converted. The speed of EXIdizer is about three times faster than EXIP in the best case. The XML candidates are very similar in performance, while there is the trend of Expat being slightly faster than Libxml2.

10.3.4 Usability Evaluation and Outlook

The proposed source code generation technique presented in this chapter highly depends on the XML schema that describes the data exchange format. Nonetheless the generated code mostly has a footprint of a few kBytes only. Section 10.3.2 introduced some numbers of an example application.

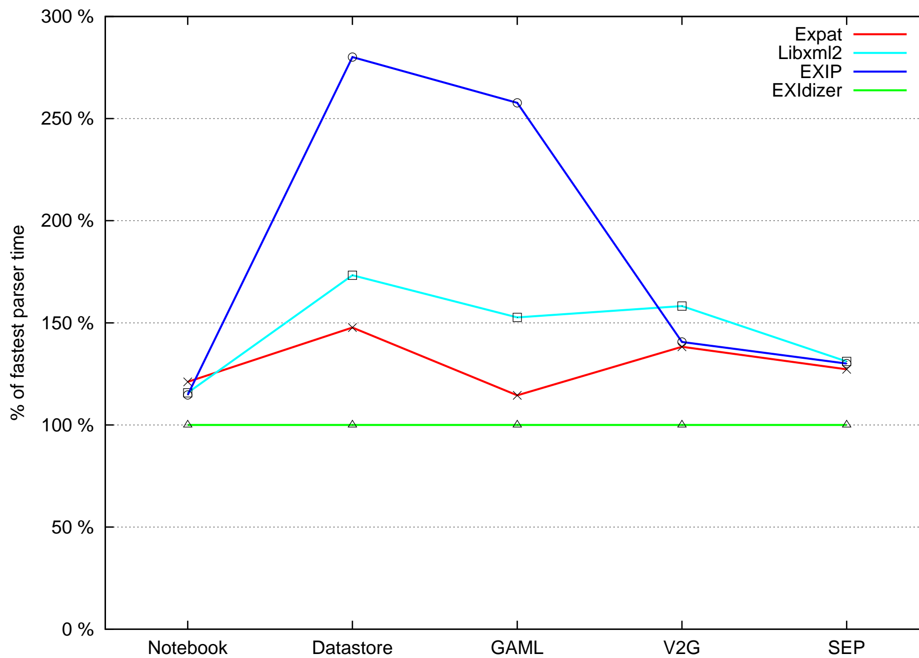


Figure 10.4: XML/EXI Parser Performance

From the perspective of an implementor (i.e., sensor board implementor), the proposed EXI solution does not implicate any burden. Implementations might not even notice that information is transmitted in an efficient XML format. The mode of operation while developing an application does not change. The only difference is that instead of building own data container classes, the ones from the generated code may be used (see Section 10.2.4). Yet another advantage over existing solutions is the flexibility of the generated code approach. Compared to hand-optimized data exchange formats, the adaptation to new circumstances, with regard to revised data structures, is very flexible and requires minimal efforts and costs. The generation process of a new EXI processor is fast and demands no knowledge about the actual EXI coding itself.

Further, the concept has been already successfully applied to projects such as OpenV2G[13] that provides an open source implementation of the ISO/IEC 15118 specification. The source code for supporting EXI streams is automatically generated by the EXIdizer tool. The same process can be applied for other relevant standards such as the Smart Energy Profile 2.0 (SEP2).

Moreover, Sebastian Käbisch presented in his thesis [47] an algorithm that allows supporting the automata (i.e., EXI grammars) for the EXI binary XML format while at the same time supporting another binary XML format, namely BiM. The idea of having the possibility to provide support for various formats

[13] OpenV2G website, `http://openv2g.sourceforge.net/`

with minimal overhead sounds very promising and may also help to even further facilitate the use of the rather new EXI technology in areas where existing formats are used today.

Further, optimization in regard to code footprint can be continued. One example is where only the encoder or the decoder part is required. The code for the relevant coding part needs to be deployed only. Moreover, often it is the case that an encoder deals with certain elements and the decoder with other elements (e.g., web services use request and response messages to communicate). In such a situation, the code can be generated for given elements only (assuming this context knowledge is available).

Chapter 11

Memory-Sensitive XML Querying

Many XML tools and XML-based applications directly or indirectly rely on the Document Object Model (DOM) [56]. This includes, for example, the XML Path Language (XPath) [17] for specifying and evaluating path expressions on XML document instances, XQuery, and XSLT processors, XML schema validators, XForms, and applications built on top of these tools.

Many XML-based processors today work on an in-memory representation of the entire XML document instance, usually represented as a DOM. The in-memory representation of a document can become very large—even larger than the size of the corresponding textual XML file in the file system (about 400% of the file size according to [50]). When the file contains a representation of the XML data in the W3C's Efficient XML Interchange (EXI) [73] format, the discrepancy between file size and in-memory size of the DOM is even larger. Especially when dealing with larger XML document instances, the potentially excessive memory consumption constitutes a problem and may render traditional DOM processing infeasible on embedded and other resource-limited devices such as cell phones and digital cameras. Nevertheless, the flexibility and extensibility of XML makes using XML and XPath desirable even on such limited devices. Consider, for example, the use of SVG [30] for creating animated user interfaces on digital cameras. Enabling such use cases requires a means for loading an XML instance into a DOM and evaluating XPath expressions in the face of restricted memory capacities. In this chapter, an approach for dynamically loading and unloading DOM elements using EXI features is presented.

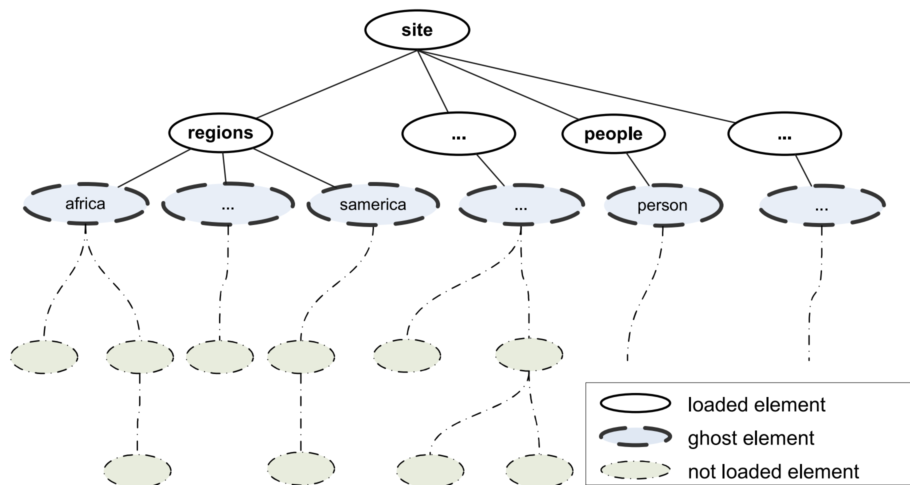


Figure 11.1: Lazy Document Object Model containing loaded and not loaded elements

11.1 Memory-Sensitive Model

The introduction shortly mentioned the idea behind the memory-sensitive model and highlights its simplicity but also its effectiveness in the sense of reducing memory consumption and decreasing processing time. In the following, the general concept will be described and the requirements of the approach will be outlined. Moreover, the technical realization is subsequently discussed also.

11.1.1 LazyDOM Concept

Lazy evaluation is a well-known technique in the area of programming languages. Delaying a computation until its result is required can increase performance by avoiding unnecessary calculations. A similar concept can be adopted to operate on an in-memory model of an XML document. The Document Object Model (DOM) is a W3C specification and can be seen as an in-memory representation of an XML information set [78], providing at any time the full information of the actual XML instance to navigate through or operate on.

An XML instance is generally transformed entirely into a DOM while multiple interactions, such as look-up operations, may tackle only certain elements of the XML tree. The possibility of working with such independent XML fragments paves the way for efficient processing and saving runtime memory. Hence, at a given time, only a fragment of the document is required. In the following, the term LazyDOM is used for referring to a DOM that partially loads XML elements and unloads elements that are not needed anymore.

Figure 11.1 depicts a fragmentation example of an XML document that represents an arbitrarily subdivided document. The root element `site` and its children

build the basic structure. The third XML level consists of ghost elements. Loaded elements are present in memory while ghost elements indicate that there is no subtree available in memory, unless it has been loaded on demand. Children of ghost elements, labeled as not loaded elements, are parsed and built only if required and can be unloaded later if necessary. This fragmentation can be done in a nested fashion, meaning that a not loaded element may contain further ghost elements.

11.1.2 LazyDOM Requirements

From the practical point of view, the LazyDOM solution has to face certain requirements to fulfill the described concept. A ghost element needs to be parsed independently from the rest of the document. To do so, the basic document needs to allow partial loading while also allowing random access on an XML element basis.

In XML terms this means that an XML instance is only partially loaded into a DOM and that currently unnecessary elements are skipped. Those skipped portions of a document (ghost elements) need to be accessible and indexable without any dependencies concerning previous data (e.g., XML namespace declarations).

The stated requirements could possibly be fulfilled using plain XML, but only with a fair amount of work and complexity. Hence the EXI format is chosen, which is identical with XML on the basis of the Information Set [79] and already offers some of the required capabilities, not to mention other beneficial characteristics of EXI such as fast processing and minimal runtime requirements.

11.2 Technical Realization

The EXI format uses a relatively simple grammar-driven approach, which achieves very efficient encodings (EXI streams) for a broad range of use cases. Due to a straightforward encoding algorithm and a small set of data types, EXI processors can be implemented on devices with limited capacity. Besides other relevant properties such as encodings with and even without schema information as well as schema deviations or partial schemas, the EXI format offers a variety of additional useful features. In this chapter, the focus is on an aspect called *selfContained* element.

In EXI terms, a *selfContained* element is a portion of an XML information set that may be read independently from the rest of the EXI document, meaning that it offers random access on XML element level. The EXI specification itself does not restrain the use of *selfContained* elements to a certain mechanism. An application is free to make use of this capability in a convenient way. The LazyDOM proposal uses this feature to realize the concept of a ghost element. Hence, both terms, ghost elements and selfContained elements, are used as synonyms throughout this elaboration.

11.2.1 Indexing Mechanism

Subsequently, a simple, yet powerful indexing mechanism is introduced. Based on the previously introduced requirements (see Section 11.1.2), a LazyDOM indexing solution needs to provide three information items:

- First of all, a selfContained element needs to be uniquely identifiable. The mechanism which is proposed is inspired by XPath [17], a language for selecting nodes in XML documents. Let us assume the following XPath expression, composed of three XPath nodes:

  `/site[1]/people[1]/person[2]`

  This expression uniquely identifies the second person node of the first people node of the first site node.

- Second, the byte offset (relative the EXI document) of a given selfContained element is needed to randomly access the element.

- Third, the length of each selfContained element needs to be known to support skipping of such elements.

Figure 11.2 depicts the outcome of the indexing format expressed in XML schema notation. Multiple indices are glued together to a set of indices, depicted as `scIndices`.

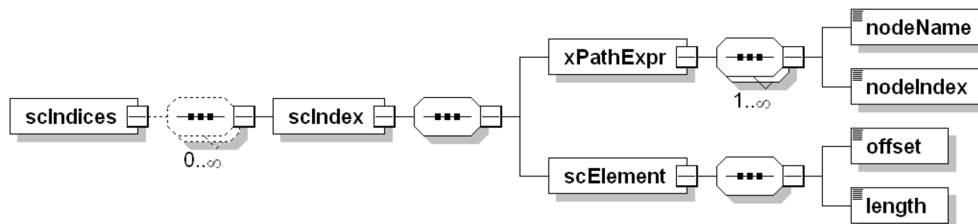


Figure 11.2: XML Schema for index structure

A selfContained element index `scIndex` is composed of an XPath expression `xPathExpr` and a selfContained element information set `scElement`. An XPath expression in turn has one to many `nodeName` and `nodeIndex` tuples while the selfContained element portion comprises the byte `offset` for random access and the selfContained element `length` in bytes.

The index example in Listing 11.1 maps exemplarily the previously used XPath expression to the byte offset `233` in an EXI document. This means that one needs to skip 233 bytes from the start of the EXI stream before reaching the relevant selfContained element person while the length of the selfContained element is `81` bytes.

Listing 11.1: Index Example Document

```
<?xml version="1.0" encoding="UTF-8"?>
<scIndices>
    <scIndex>
        <xPathExpr>
            <nodeName>site</nodeName>
            <nodeIndex>1</nodeIndex>
            <nodeName>people</nodeName>
            <nodeIndex>1</nodeIndex>
            <nodeName>person</nodeName>
            <nodeIndex>2</nodeIndex>
        </xPathExpr>
        <scElement>
            <offset>233</offset>
            <length>81</length>
        </scElement>
    </scIndex>
    <!-- more scIndex elements -->
</scIndices>
```

The use of XML technologies (e.g., XML schema) to describe the index has the advantage of being able to use the EXI format to efficiently store the index in an easily consumable way without introducing new requirements or technologies. EXI uses local names to keep the overhead at a minimum. Instead of coding the element name as a character sequence over and over again, EXI uses compact identifiers that are already part of EXI string tables.

11.2.2 LazyDOM Applicability

As stated in the introduction, the goal is to offer the same functionality as any other Document Object Model realization, with the benefit of having just the relevant parts of an XML tree in memory.

Hence the proposed LazyDOM solution uses the described indexing mechanism and initially forms a skeletal structure, meaning that the root element and the entire substructure up to any indexed element is resolved and loaded into memory (see Figure 11.1).

An XPath engine or any application using a DOM does not see any difference to a conventional DOM implementation. A DOM node or element has to provide capabilities for navigating the tree. Hence, each node has to provide a link to the parent, the previous and the next sibling, as well as a complete list of child nodes. That said, the LazyDOM solution must provide exactly the same functionality required by the W3C DOM specification without any restrictions.

11.3 Results and Discussion

This section provides test results showing that the memory-sensitive approach presented in this chapter works well for real test data. For comparison with other projects, such as Projecting XML Documents [50], as well as for traceability, test-data generated within the XMark Benchmark Project[1] is used. XMark [72] is an XML Benchmark Project which aims to provide a benchmark suite that allows users and developers to gain insights into the characteristics of their XML repositories. In addition, Java's Management Extensions (JMX)[2] is used, a Java technology that supplies tools for managing and monitoring Java applications, e.g., in terms of memory consumption.

11.3.1 Test Data

XMark provides a data generator that has been used to produce XML documents with sizes varying from 1 to 116 MB (e.g., the document xmark-f-0-10.xml was produced using the XMark factor 0.10, see Table 11.1).

Table 11.1: XMark test documents

TestCase \ kB	**XML**	**EXI**	**Idx**	(#)
xmark-f-0-01.xml	1 155	812	7	(498)
xmark-f-0-10.xml	11 597	8 076	72	(4931)
xmark-f-1-00.xml	115 775	79 133	731	(49256)

The comparison is split into two steps. First of all, EXI encoded streams are produced, which constitute counterparts to the XMark generated textual XML documents. Second, based on those generated XML/EXI documents, several XPath queries are applied and tested.

Table 11.1 shows the different documents and their sizes on disk. EXI stands for the EXI-encoded document with selfContained elements. We decided to introduce selfContained elements for all elements appearing on the third XML tree level. The last column Idx shows the overhead of the index structure by mentioning the size of the index and quoting the number of indices (#).

It should further be noted that the size of the index structure compared to the documents itself is less than 1%. This seems like a reasonable overhead in terms of storage and parsing requirements. Please note also that EXI documents are usually about ten or more times smaller than semantically equivalent XML documents (see EXI Evaluation [9]).

Due to the fact that the XMark generator produces instances with string datatypes only and the XML structure is minimal compared to the actual content

[1] `http://www.xml-benchmark.org/`
[2] `http://java.sun.com/products/JavaManagement/`

data, the EXI format can not properly show its expected benefit. Nevertheless, this chapter is not about showing EXI's compression efficiency. Instead it focuses on a small subset of EXI format features that facilitate building the LazyDOM solution.

11.3.2 Query Set

To show the effectiveness of the LazyDOM approach, three XPath queries are introduced. These serve as a basis for the subsequently presented measurements regarding memory consumption and processing time.

Query Q1 $/site/people/person[2]/name$
describes a very precise request looking for a person's name with a position index.

Query Q2 $/site/regions/*$
returns a node list of regions such as `asia` and `europe`.

Query Q3a $//closed_auction[date =' 04/16/2000']$
tackles essentially the entire document. The request is stated in such a generic way (// is short for /descendant-or-self::node()) that an XPath engine is required to traverse the entire XML tree and filter `closed_auction`'s with a certain date.

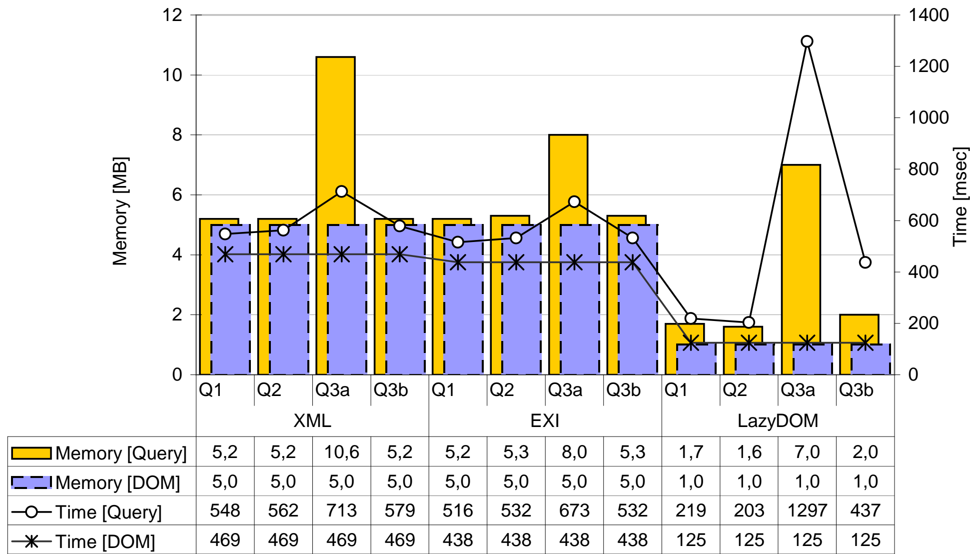
Query Q3b $/site/auctions/closed_auction[date =' 04/16/2000']$
is semantically equivalent to Query Q3a according to the XMark schema. The improvement is though that the request uses absolute paths to identify the node. We will get back to the significant difference between Q3a and Q3b further below.

11.3.3 Performance Measurements

The testbed is composed of a notebook running Windows XP with 1.66 GHz and 2 GB of RAM. We used the Java Virtual Machine 1.6 (with default options). From the many possible XPath engines, the widely used and well established Jaxen[3] engine has been selected.

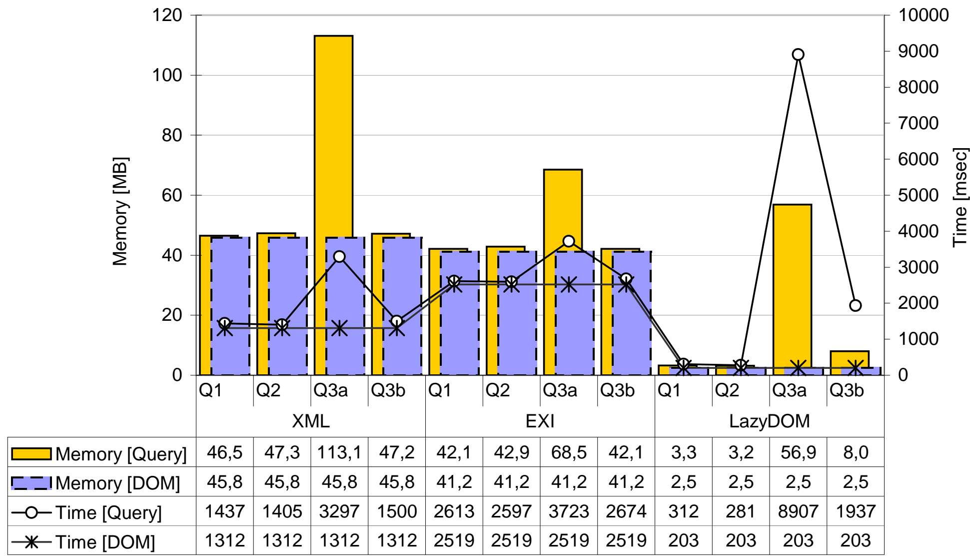
Figure 11.3 shows a digest of the memory-consumption measurements and the according processing times. The bar diagram presents the peak size in MB of the Java heap during execution (e.g., the JVM option `-Xmx128m` sets the maximum heap size to 128 MB) while the line diagram respectively shows query loading/execution times in milliseconds.

[3] `http://jaxen.codehaus.org/`



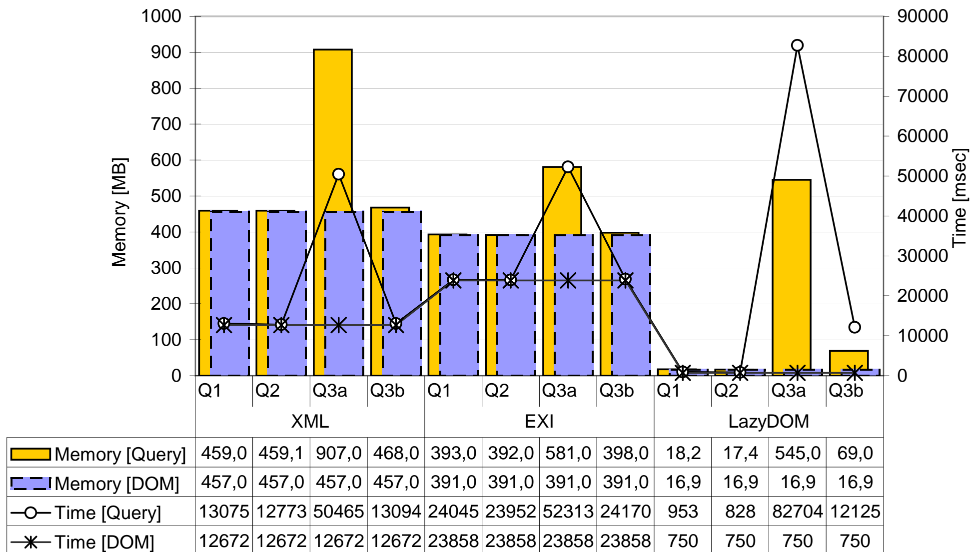
	XML Q1	XML Q2	XML Q3a	XML Q3b	EXI Q1	EXI Q2	EXI Q3a	EXI Q3b	LazyDOM Q1	LazyDOM Q2	LazyDOM Q3a	LazyDOM Q3b
Memory [Query]	5,2	5,2	10,6	5,2	5,2	5,3	8,0	5,3	1,7	1,6	7,0	2,0
Memory [DOM]	5,0	5,0	5,0	5,0	5,0	5,0	5,0	5,0	1,0	1,0	1,0	1,0
Time [Query]	548	562	713	579	516	532	673	532	219	203	1297	437
Time [DOM]	469	469	469	469	438	438	438	438	125	125	125	125

(a) xmark-f-0-01.xml (ca. 1 MB of XML)



	XML Q1	XML Q2	XML Q3a	XML Q3b	EXI Q1	EXI Q2	EXI Q3a	EXI Q3b	LazyDOM Q1	LazyDOM Q2	LazyDOM Q3a	LazyDOM Q3b
Memory [Query]	46,5	47,3	113,1	47,2	42,1	42,9	68,5	42,1	3,3	3,2	56,9	8,0
Memory [DOM]	45,8	45,8	45,8	45,8	41,2	41,2	41,2	41,2	2,5	2,5	2,5	2,5
Time [Query]	1437	1405	3297	1500	2613	2597	3723	2674	312	281	8907	1937
Time [DOM]	1312	1312	1312	1312	2519	2519	2519	2519	203	203	203	203

(b) xmark-f-0-10.xml (ca. 12 MB of XML)



	XML Q1	XML Q2	XML Q3a	XML Q3b	EXI Q1	EXI Q2	EXI Q3a	EXI Q3b	LazyDOM Q1	LazyDOM Q2	LazyDOM Q3a	LazyDOM Q3b
Memory [Query]	459,0	459,1	907,0	468,0	393,0	392,0	581,0	398,0	18,2	17,4	545,0	69,0
Memory [DOM]	457,0	457,0	457,0	457,0	391,0	391,0	391,0	391,0	16,9	16,9	16,9	16,9
Time [Query]	13075	12773	50465	13094	24045	23952	52313	24170	953	828	82704	12125
Time [DOM]	12672	12672	12672	12672	23858	23858	23858	23858	750	750	750	750

(c) xmark-f-1-00.xml (ca. 116 MB of XML)

Figure 11.3: Memory consumption and query execution times

The set of queries (Q1, Q2, Q3a and Q3b) groups the three candidates, XML, EXI, and LazyDOM. XML designates parsing and loading the DOM from a textual XML document. EXI in turn parses and loads an EXI document. The LazyDOM, in contrary, parses an EXI document but only loads required portions of the document into the DOM.

To demonstrate the difference between initially loading the XML information set to a DOM and the additional overhead for processing queries, both measurements were split. Figure 11.3 shows memory consumption (Memory) and processing times (Time) for both, DOM loading only (DOM) and DOM loading together with subsequent query execution (Query). Three XMark generated XML test documents (a), (b), and (c) varying in size from 1 MB to 116 MB demonstrate the applicability of the LazyDOM in diverse memory magnitudes.

Memory Consumption

Figure 11.3 illustrates that the memory consumption and processing times for DOM loading is similar for XML and EXI. The reason for the slight difference in memory size is due to the fact that the EXI format prunes insignificant white spaces.

The proposed LazyDOM solution shows a huge memory benefit especially for queries Q1, Q2, and Q3b, where only a portion of the entire document is of interest. If we deal with inefficient XPath queries (e.g., Q3a `//closed_auction [date='04/16/2000']`) the entire tree has to be visited. An approach has been implemented that unloads *selfContained* elements that are no longer required if memory becomes scarce. It keeps track of a list of least recently used *selfContained* elements and removes the least recently used *selfContained* element each time an additional element is to be loaded and the number of loaded elements has reached a configurable number N (e.g., in the test cases $N = 10$). Nevertheless, the gain compared to conventional engines is not as outstanding as expected (see query Q3a in Figure 11.3). The reason is the implemented caching strategy of many XPath engines. The less an XPath engine makes use of caches, the more memory can be freed by Java's garbage collector.

Processing Time

Loading the information set into a DOM shows advantages for the LazyDOM given that initially only a small amount of data needs to be parsed. In the second measurement step, query processing takes effect. The additional querying process does not indicate any noteworthy processing time difference between the test candidates XML and EXI. This matches the expectations given that there is no difference between a DOM created from an XML or an EXI document.

In terms of processing efficiency, the LazyDOM shows improved runtime performance for XPath expressions requiring the loading or unloading of no or only

few *selfContained* elements. When many load or unload operations take place, e. g., because of heavily restricted memory availability, processing efficiency suffers. However, evaluating the query will still succeed instead of failing with an out-of-memory error.

The measurements show that the developed solution represents a well-behaved DOM strategy, loading sub-elements of a document only when the actual need occurs. In any case, the overhead is kept at a minimum. Furthermore, both the granularity of indexed *selfContained* elements in EXI as well as the number N of buffered *selfContained* elements are tunable. Hence, it offers a *transparent* mechanism to reduce the overall memory consumption for devices with limited resources as well as for powerful servers.

11.3.4 Design Guidelines

The LazyDOM offers a tunable means to potentially reduce the memory consumption and at the same time improve the performance of DOM-based XML processing. As with many optimization techniques, the amount of improvement that can be achieved depends on the actual use case. The configuration of the LazyDOM and the components interacting with it should be carefully tailored to the demands of the actual application. In this section, some design guidelines are described that illustrate how LazyDOM could and should be used in practice to achieve best results.

LazyDOM: Where and When

It should be pointed out that there are use cases where LazyDOM might not be the best choice. In an environment that is always able to load the entire DOM into memory and that always accesses all or at least most of the data in the DOM, dynamically loading the DOM has little or no benefit in terms of memory consumption but can incur a performance penalty. Consider Query Q3a in Figure 11.3 as an example. Since the query uses the XPath descendant-or-self axis, all of the DOM is traversed during query evaluation. Thus, the savings in memory consumption when using LazyDOM is limited compared to a traditional DOM. But the processing time is increased due to the overhead of indexing and dynamic loading.

However, if only parts of the DOM are actually needed, dynamically loading only these parts will not only save memory but will also lead to faster DOM loading since much less data needs to be loaded overall. This can be seen, for example, from the performance results for Query Q3b in Figure 11.3. The query avoids using the XPath descendant-or-self axis and directly queries only parts of the DOM. Thus, memory consumption and processing time are heavily decreased.

Furthermore, in an environment that is not able to always load the entire DOM into memory due to memory restrictions, LazyDOM is enabling the use of DOM-based XML processing in the first place. If, in this case, only parts of the DOM are actually required by an application, the reduced memory consumption is again combined with improved processing performance as described above. If all or most of the data in the DOM is referenced by an application, processing performance can be worse than loading the entire DOM into memory upfront since repeated loading and unloading of *selfContained* elements will occur. Still, since loading the entire DOM into memory is not possible in memory restricted environments, LazyDOM enables the use of DOM-based XML processing at the cost of potentially increased processing times.

Granularity of selfContained Elements

A major means of configuring the LazyDOM is the choice of granularity of *selfContained* elements. Identifying more and smaller *selfContained* elements—e.g., at lower levels of the XML tree—allows more fine-grained control over memory consumption since smaller parts of the DOM can be loaded and unloaded. On the other hand, this approach increases indexing overhead and reduces EXI compression.

Choosing less and larger *selfContained* elements—e.g., at higher levels of the XML tree—leads to less fine-grained control over memory consumption but reduces indexing overhead and improves EXI compression.

11.3.5 Related Work

Busatto et al. [13] presented a technique that represents the tree structure of an XML document in an efficient way by exploiting the high regularity of XML documents. Repetitions of tree patterns are detected and removed. The used technique is a generalization of the approach of sharing common subtrees, if a subtree has occurred before it is represented by a pointer to its previous occurrence. This approach is orthogonal to LazyDOM. Combining both solutions, LazyDOM and the efficient representation of the XML tree structure, would seem to reduce overall memory consumption even further.

The approach of Kim et al. [41] relates to the previously presented technique in the sense that large XML documents are partitioned into smaller XML trees. When retrieval operations take place, in contrast, a unify operation is required to unify split child nodes.

The way DOM-based applications, such as XPath, XQuery, or XSLT processors, interact with the DOM can have significant impacts on the improvements that are achievable by combining these applications with LazyDOM. One example to be mentioned here is the concept of schema-aware XPath processors (e.g.,

Paparizos et al. [64]). Considering Query Q3a in Section 11.3.2, a non-schema-aware XPath processor has to traverse all of the DOM to answer the query. Thus, the entire DOM tree needs to be loaded (and potentially unloaded) during the process. Query Q3b uses schema knowledge to eliminate the descendant-or-self axis in the XPath expression of Query Q3a to obtain the same result without having to traverse all of the DOM.

Many efforts have been made to reduce the memory requirements of XQuery and XPath processors. An approach for reducing the memory requirements has been implemented in the Galax XQuery processor [50]. This approach is based on an a priori analysis of the query to be evaluated. Only the parts of the document needed to correctly and completely process the query are subsequently loaded into an in-memory DOM. Depending on the query, this might require significantly less memory than loading the entire document. However, the analysis process needs to be repeated and the identified necessary parts of the XML document need to be reloaded for each new query. In contrast, the LazyDOM can use the same DOM for evaluating each XPath expression referencing the corresponding XML document. Also, when dealing with queries that need most of or even the entire document during processing, e. g., queries using the descendant-or-self axis on the document root, the a priori analysis of the query yields little or no benefit since all necessary parts of the document need to be loaded into memory as a whole. The solution presented in this chapter allows to dynamically load and unload DOM subtrees during XPath evaluation. Hence, even if the parts of the document necessary for evaluating a certain XPath expression do not fit into memory as a whole, it is still possible to correctly process the document within the available memory boundaries.

Streaming Transformations for XML (STX) [16] allow the transformation of large, theoretically infinite XML documents or XML data streams with bounded memory requirements. STX processes the XML data in a streaming fashion. The limitation of memory consumption during the processing of XPath expressions results from a limitation of the allowed XPath expressions to a subset of XPath that is suitable for streaming evaluation. Thus, the evaluation does not require the buffering of a possibly infinitely large internal state. Compared to STX, the presented approach is aimed at supporting the full functionality of any DOM-based application.

Part IV

Résumé

Chapter 12

Contributions

At the time of writing, the co-developed Efficient XML Interchange (EXI) format has been starting to be successfully applied in various domains. The ZigBee Smart Energy Profile 2.0 specification [7] is one example. Another interesting area is the Vehicle 2 Grid Communication Interface (V2G CI) specification, which is elaborated and known under ISO/IEC 15118 [3, 4]. Both mentioned specifications make use of the EXI technology and are facing similar issues such as runtime constraints in regard to processing power and memory consumption. These issues have been resolved, among other efforts, due to the proposed techniques presented in this thesis. This provides evidence that it is possible to realize a demand-tailored EXI processor (see Chapter 10) that leads to the de-facto standard implementation of ISO/IEC 15118 known as OpenV2G[1]. Hence, very restricted devices can be supported in regard to processing and also in regard to memory requirements.

The editing of the EXI Primer [66] is another important work to facilitate EXI dissemination. The EXI recommendation tends to be extremely difficult to read due to its complexity and its dependency on XML schema. On the contrary, the suggested starting point for EXI readers, the Primer document, is much easier to understand and oriented towards quickly understanding how the EXI format can be used.

Moreover, the specification of the EXI Profile [29], which was elaborated in collaboration with Youenn Fablet, is dedicated to device classes and use cases that are not capable or allowed to require unpredictable memory growth at runtime. Though EXI provides a number of format options that help to constrain runtime memory usage, there are aspects of memory use that are left open in the EXI format. These aspects are examined in the EXI Profile specification and the outcome is in the progress of being published as a W3C recommendation. It is

[1] OpenV2G project website: `http://openv2g.sourceforge.net/`

important to note that the EXI Profile document specifies rules to ensure that the memory restrictions are respected while keeping compatibility with the EXI 1.0 specification [73, 29].

Also, personal effort as part of the W3C working group and as Canonical EXI [48] editor provide intrinsic support for EXI in the area of XML security. Although EXI supports *traditional* XML Signature by preserving XML information such as comments and prefixes (see EXI Best Practices for XML Signature [20]), this strategy is not suited for all environments and use cases. Hence, supporting XML/EXI canonicalization and signature without going through plain-text XML where nothing else but EXI is available is needed. Canonical EXI establishes a method for determining whether two EXI documents are equivalent, for the purposes of applications, while differing in physical representation.

My achievements presented in this thesis, both in theoretical and practical regards, can be summarized as follows:

- Development of the de-facto reference implementation (see EXIficient in Chapter 7) which in addition is the first and so far only open source implementation supporting all EXI features.
- Elaboration and validation of an optimized datatype representation based on the well-defined and well-studied Backus-Naur Form (Chapter 8).
- Elaboration and analysis of an EXI grammar exchange format with minimal runtime requirements but still being very efficient on the wire (Chapter 9).
- Formulation of a grammatically sound and automatable process for generating demand-tailored EXI processors (Chapter 10).
- Elaboration of a memory-sensitive in-memory representation to keep memory (RAM) requirements at the very minimum (Chapter 11).
- Editing of W3C specifications such as the EXI format [73], the EXI Primer [66], the EXI Profile [29], and Canonical EXI [48]. Relevant publications on related topics.

The EXI work has been completed. However, whenever new application fields appear, new requirements emerge that may demand further elaborations. Such an example is the Extensible Messaging and Presence Protocol (XMPP). The protocol was originally named Jabber and is used for publish&subscribe systems, signalling for VoIP, video, file transfer, gaming, Internet of Things applications such as the smart grid, and social networking services. Most of the requirements in that area are provided by the EXI technology and the work that has been

evolved around the technology already. That said, a *new* requirement that is currently being discussed is the aspect of XML schema or respectively EXI grammar negotiation. XMPP is highly flexible in regard to extensions that are in use and hence demands the same flexibility from the transport protocol. In this connection, the goal is to identify which schema knowledge is usable and used to represent the data. Moreover, also theoretically changing the set of information while processing an EXI stream is considered to be useful. Moreover, flushing of pending events in a streaming scenario is another use case. So far EXI pit-packed mode may encounter pending bits[2] that prohibit streaming in some rare cases due to the missing possibility of flushing the stream. The possibility of introducing a flushing mechanism is also analyzed.

Another interesting aspect that is often arising when data exchange in the Web is discussed is how JSON data (a text-based human-readable data interchange derived from the JavaScript scripting language) support in conjunction with EXI can be improved.

Finally, it is worth to note that the elaborated EXI technology seems to head towards a bright future and dissemination has been taking place in various areas.

[2]We speak about pending bits in EXI if the information does not fully fill up all bits of the according byte. That said, unless the byte is not complete it cannot be sent.

Chapter 13

Conclusions

The Efficient XML Interchange (EXI) format has, since its publication as a W3C Recommendation in March 2011, reached a very decent acceptance and dissemination in the XML community. Also, there has been a lot of interest and commitment from other standardization bodies. For example, EXI has been officially selected as the interchange format for the ISO/IEC 15118 [4] specification and the DIN 70121 [2]. Further, the ZigBee Smart Energy Profile 2.0 Application Protocol [7] is another specification where EXI is used. Yet another application is the Extensible Messaging and Presence Protocol (XMPP) which is a communication protocol for message-oriented middleware based on XML, originally named Jabber, also planning to include EXI support[1].

Hence, efforts within the W3C community and also the interest from other standardization bodies confirm on the one hand the effectiveness of the proposed format. On the other hand, it shows the actual need for such an efficient exchange protocol that fits the real word requirements in the first place. It turns out that the Efficient XML Interchange specification, its improvements, and the manifold applications that have been elaborated in this thesis provide to the XML community what they were expecting for such a long time.

Also in the area of available implementations, a lot has happened since the beginning of the first efforts. Publicly available commercial implementations and also freely available implementations in various programming languages (e.g., Java, C/C++, C# and .NET) have been constructed and deployed[2]. More than that, also numerous internal implementations can be assumed just by following the activities of relevant companies.

[1] Request on public EXI mailing list, `http://lists.w3.org/Archives/Public/public-exi/2013Mar/0000.html`

[2] List of EXI implementations available at: `http://www.w3.org/XML/EXI/#implementations`

One example that shows the overall interest very well is the absolute number of downloads since its first publication of the de-facto reference EXI processor EXIficient, which reached almost ten thousand downloads in April 2014. Moreover, external tools around EXIficient have been built for teaching[3] and other purposes too. Personally speaking we do expect many interesting projects and efforts to follow.

The EXI optimizations of this thesis that have been theoretically elaborated in the first place and subsequently practically validated are also of huge importance. Some proposed solutions and techniques may even become standardized in a future EXI version such as the datatype representation elaborated in Chapter 8 or the grammar representation introduced in Chapter 9. Chapter 10 and Chapter 11 are more heading towards how implementations can be built to work with a minimum amount of memory and processing power and are already used in real-world applications such as the Siemens charging spot for electrical vehicles.

[3]ExiProcessor is for example a command-line demonstration program that uses the EXIficient library and allows encoding text XML files into binary EXI and vice-versa again, `http://sourceforge.net/p/exiprocessor/`

Part V

Appendices

Bibliography

[1] ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF, 1996. [cited at p. 79, 84]

[2] Electromobility - Digital communication between a DC EV charging station and an electric vehicle for control of DC charging in the Combined Charging System, 2012. http://www.din.de. [cited at p. 4, 157]

[3] ISO 15118-1:2013 Road vehicles – Vehicle to grid communication interface – Part 1: General information and use-case definition, 2013. [cited at p. 4, 52, 134, 153]

[4] ISO/FDIS 15118-2 Road vehicles – Vehicle to grid communication interface – Part 2: Network and application protocol requirements, 2013. [cited at p. 4, 52, 153, 157]

[5] ISO/IEC 14496-20. Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF), December 2008. 2nd Edition. [cited at p. 29, 74, 88]

[6] Serge Abiteboul. Querying Semi-Structured Data. In Foto N. Afrati and Phokion G. Kolaitis, editors, *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1997. [cited at p. 8]

[7] ZigBee Alliance and HomePlug Powerline Alliance liaison. Smart Energy Profile 2.0 Application Protocol Specification. Draft, July 2012. 0.9 Standard, http://www.zigbee.org/Standards/ZigBeeSmartEnergy/Overview.aspx. [cited at p. 4, 52, 134, 153, 157]

[8] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. ALGOL-like Languages, Volume 1. chapter Revised Report on the Algorithmic Language ALGOL 60, pages 19–49. Birkhauser Boston Inc., Cambridge, MA, USA, 1997. [cited at p. 78]

[9] Carine Bournez. Efficient XML interchange evaluation. W3C working draft, W3C, April 2009. http://www.w3.org/TR/2009/WD-exi-evaluation-20090407. [cited at p. 76, 144]

[10] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. [cited at p. 16]

[11] Tim Bray, François Yergeau, C. M. Sperberg-McQueen, Jean Paoli, and Eve Maler. Extensible Markup Language (XML) 1.0 (Fourth Edition). Technical report, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816. [cited at p. 15]

[12] David Brownell and David Megginson. Simple API for XML (SAX). Version 2.0.2, http://www.saxproject.org. [cited at p. 53, 120]

[13] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML documents. In *10th International Symposium, Database Programming Languages, Trondheim, Norway*, pages 199–216. Springer, August 2005. [cited at p. 149]

[14] Tolga Capin. Mobile SVG Profiles: SVG Tiny and SVG Basic. W3C Recommendation, W3C, January 2003. http://www.w3.org/TR/2003/REC-SVGMobile-20030114/. [cited at p. 74]

[15] Noam Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory IT-2*, 2(3):113–124, September 1956. [cited at p. 78]

[16] Petr Cimprich, Oliver Becker, Christian Nentwich, Honza Jiroušek, Manos Batsis, Paul Brown, and Michael Kay. Streaming Transformations for XML (STX) Version 1.0. `http://stx.sourceforge.net/documents/spec-stx-20070427.html`, 2007. Working Draft. [cited at p. 150]

[17] James Clark and Steven DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116. [cited at p. 139, 142]

[18] Mike Cokus and Santiago Pericas-Geertsen. XML Binary Characterization Properties. W3C Note, W3C, March 2005. http://www.w3.org/TR/2005/NOTE-xbc-properties-20050331/. [cited at p. 3, 30]

[19] Mike Cokus and Santiago Pericas-Geertsen. XML Binary Characterization Use Cases. W3C Note, W3C, March 2005. http://www.w3.org/TR/2005/NOTE-xbc-use-cases-20050331/. [cited at p. 3, 30]

[20] Mike Cokus and Daniel Vogelheim. Efficient XML Interchange Best Practices. W3C Working Draft, W3C, December 2007. http://www.w3.org/TR/2007/WD-exi-best-practices-20071219/. [cited at p. 154]

[21] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 4234 (Draft Standard), October 2005. Obsoleted by RFC 5234. [cited at p. 79]

[22] Peter J. Denning, Jack B. Jack Bonnell Dennis, and Joseph E. Qualitz. *Machines, languages, and computation.* Englewood Cliffs, N.J. Prentice-Hall, 1978. [cited at p. 78]

[23] Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3 (IETF RFC RFC1951). An online version is available at `http://www.ietf.org/rfc/rfc1951.txt`, May 1996. [cited at p. 25, 110]

[24] J. Dufourd, O. Avaro, and C. Concolato. LASeR: the MPEG Standard for Rich Media Services, 2010. Whitepaper, http://www.mpeg-laser.org/documents/LASeRWhitePaper.pdf. [cited at p. 29, 74, 88]

[25] ETSI. Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime")—Part 2: Phase 1 - System description. Technical Report ETSI TS 102 822-2, Union Européenne de Radio-Télévision, 2007. [cited at p. 30]

[26] ETSI. Digital Video Broadcasting (DVB)—Carriage of Broadband Content Guide (BCG) information over Internet Protocol (IP). Technical Report ETSI TS 102 539, Union Européenne de Radio-Télévision, 2010. [cited at p. 30]

[27] ETSI. Digital Video Broadcasting (DVB)—IP Datacast over DVB-H: Electronic Service Guide (ESG). Technical Report ETSI TS 102 471, Union Européenne de Radio-Télévision, 2010. [cited at p. 30]

[28] ETSI. Digital Video Broadcasting (DVB)—Carriage and signalling of TV-Anytime information in DVB transport streams. Technical Report ETSI TS 102 323, Union Européenne de Radio-Télévision, 2012. [cited at p. 30]

[29] Youenn Fablet and Daniel Peintner. Efficient XML Interchange (EXI) Profile. W3C working draft, W3C, July 2012. http://www.w3.org/TR/2012/WD-exi-profile-20120731/. [cited at p. 153, 154]

[30] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. W3C recommendation, W3C, January 2003. http://www.w3.org/TR/2003/REC-SVG11-20030114/. [cited at p. 74, 81, 139]

[31] Christopher Fry and Deva Sagar. JSR 173: Streaming API for XML (StAX). Java Specification Request 173, Version 2.9, https://jcp.org/en/jsr/detail?id=173. [cited at p. 53, 121]

[32] Neil Gershenfeld. *When things start to think.* Coronet Books Hodder & Stoughton, 1999. [cited at p. 3]

[33] Oliver Goldman and Dmitry Lenkov. XML Binary Characterization. W3C Note, W3C, March 2005. http://www.w3.org/TR/2005/NOTE-xbc-characterization-20050331/. [cited at p. 30]

[34] Sheila A. Greibach. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM*, 12:42–52, January 1965. [cited at p. 80]

[35] Peter Haggar and Stephen D. Williams. XML Binary Characterization Measurement Methodologies. W3C Note, W3C, March 2005. http://www.w3.org/TR/2005/NOTE-xbc-measurement-20050331/. [cited at p. 30]

[36] J. Heuer, C. Thienot, and M. Wollborn. 2.3 MEPG-7 Binary Format. In Thomas Sikora B. S. Manjunath, Philippe Salembier, editor, *Introduction to MPEG-7: Multimedia Content Description Interface*. Wiley, 2002. [cited at p. 29, 91]

[37] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Cambridge, 1979. [cited at p. 78]

[38] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952. [cited at p. 25, 36]

[39] ISO. Information technology Multimedia framework (MPEG-21)—Part 16: Binary Format. ISO ISO/IEC 21000-16 ed1.0, International Organization for Standardization, 2005. [cited at p. 29]

[40] Jaakko Kangasharju. *XML Messaging for Mobile Devices*. PhD thesis, University of Helsinki, Faculty of Science, Department of Computer Science, 2008. [cited at p. 8, 25]

[41] Seung Min Kim, Suk I. Yoo, Eunji Hong, Tae Gwon Kim, and Il Kon Kim. A document object modeling method to retrieve data from a very large XML document. In *ACM Symposium on Document Engineering*, pages 59–68, 2007. [cited at p. 149]

[42] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, and Abraham Heifets. XML screamer: An integrated approach to high performance XML parsing, validation and deserialization. In *15th International World Wide Web Conference*, pages 93–102. ACM Press, 2006. [cited at p. 20]

[43] R. Kurki-Suonio. Notes on top-down languages. *BIT Numerical Mathematics*, 9:225–238, 1969. 10.1007/BF01946814. [cited at p. 80]

[44] Thomas Kurz. Efficient XML Communication for Rich Internet Applications. Diploma Thesis, University of Passau, Chair of Distributed Information Systems, Germany, 9 2008. [cited at p. 72, 88]

[45] R. Kyusakov, J. Eliasson, and J. Delsing. Efficient structured data processing for web service enabled shop floor devices. In *IEEE International Symposium on Industrial Electronics (ISIE)*, pages 1716 –1721, June 2011. [cited at p. 132]

[46] S. Käbisch, D. Peintner, J. Heuer, and H. Kosch. Efficient and Flexible XML-Based Data-Exchange in Microcontroller-Based Sensor Actor Networks. In *IEEE Advanced Information Networking and Applications Workshops (WAINA), 24th International Conference*, pages 508 –513, April 2010. [cited at p. 131]

[47] Sebastian Käbisch. Generic Automaton Construction for Code Generation of Binary XML Codecs. Diploma Thesis, University of Passau, Chair of Distributed Information Systems, Germany, 11 2008. [cited at p. 137]

[48] Sebastian Käbisch and Daniel Peintner. Canonical EXI. W3C Working Draft, W3C, March 2013. http://www.w3.org/TR/canonical-exi/. [cited at p. 154]

[49] Hartmut Liefke and Dan Suciu. XMill: an Efficient Compressor for XML Data. In *SIGMOD Conference*, pages 153–164. ACM, 1999. [cited at p. 27, 41]

[50] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 213–224, Berlin, Germany, September 2003. [cited at p. 139, 144, 150]

[51] Bruce Martin and Bashar Jano. WAP Binary XML Content Format. W3C Note, W3C, June 1999. http://www.w3.org/1999/06/NOTE-wbxml-19990624/. [cited at p. 28]

[52] J. M. Martinez. MPEG-7 Overview. Technical report, 2004. MPEG-7 Overview (version 10), ISO/IEC JTC1/SC29/WG11N6828, http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm. [cited at p. 29, 41]

[53] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 242–259. Springer, 2010. [cited at p. 3]

[54] Muhammad Ali Mazidi, Janice Gillispie Mazidi, and Rolin D McKinlay. *The 8051 microcontroller and embedded systems: using Assembly and C*. Pearson/Prentice Hall, 2006. [cited at p. 5, 6, 131]

[55] Noah Mendelsohn, Anish Karmarkar, Henrik Frystyk Nielsen, Jean-Jacques Moreau, Marc Hadley, Martin Gudgin, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. [cited at p. 8]

[56] Gavin Nicol, Mike Champion, Philippe Le Hégaret, Jonathan Robie, Lauren Wood, Arnaud Le Hors, and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407. [cited at p. 53, 120, 139]

[57] U. Niedermeier, J. Heuer, A. Hutter, and W. Stechele. MPEG-7 Binary Format for XML Data. In *Proceedings of the Data Compression Conference*, DCC '02, pages 467–471, Washington, DC, USA, 2002. IEEE Computer Society. [cited at p. 29]

[58] Ulrich Niedermeier. *Algorithmus und Softwarearchitektur für die binäre Codierung von strukturierten Dokumenten*. PhD thesis, 2004. [cited at p. 91, 92]

[59] Telecommunication Standardization Sector of ITU. X.680: Information technology Abstract Syntax Notation One (ASN.1): Specification of basic notation. Recommendation, ITU-T, November 2008. http://www.itu.int/rec/T-REC-X.680/en. [cited at p. 28]

[60] Telecommunication Standardization Sector of ITU. X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Recommendation, ITU-T, November 2008. http://www.itu.int/rec/T-REC-X.690/en. [cited at p. 28]

[61] Telecommunication Standardization Sector of ITU. X.691 : Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). Recommendation, ITU-T, November 2008. http://www.itu.int/rec/T-REC-X.691/en. [cited at p. 28]

[62] Telecommunication Standardization Sector of ITU. X.693 : Information technology - ASN.1 encoding rules: XML Encoding Rules (XER). Recommendation, ITU-T, November 2008. http://www.itu.int/rec/T-REC-X.693/en. [cited at p. 28]

[63] Telecommunication Standardization Sector of ITU. X.694 : Information technology - ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1. Recommendation, ITU-T, November 2008. http://www.itu.int/rec/T-REC-X.694/en. [cited at p. 28]

[64] Stelios Paparizos, Jignesh M. Patel, and H. V. Jagadish. SIGOPT: Using Schema to Optimize XML Query Processing. In *International Conference on Data Engineering (ICDE)*, pages 1456–1460, 2007. [cited at p. 150]

[65] Daniel Peintner, Harald Kosch, and Jörg Heuer. Efficient XML interchange for Rich Internet Applications. In *Proceedings of the 2009 IEEE international conference on Multimedia and Expo*, ICME'09, pages 149–152, Piscataway, NJ, USA, 2009. IEEE Press. [cited at p. 72]

[66] Daniel Peintner and Santiago Pericas-Geertsen. Efficient XML Interchange (EXI) Primer. W3C Working Draft, W3C, December 2009. http://www.w3.org/TR/2009/WD-exi-primer-20091208/. [cited at p. 33, 42, 44, 153, 154]

[67] David Peterson, Shudi (Sandy) Gao, Paul V. Biron, Ashok Malhotra, Henry S. Thompson, Ashok Malhotra, and C. M. Sperberg-McQueen. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. Last Call WD, W3C, December 2009. http://www.w3.org/TR/2009/WD-xmlschema11-2-20091203/. [cited at p. vi, 17, 72]

[68] Mustafa H. Qureshi and M. H. Samadzadeh. Determining the Complexity of XML Documents. *International Conference on Information Technology: Coding and Computing*, 2:416–421, 2005. [cited at p. 20]

[69] P. Sandoz, A. Triglia, and S. Pericas-Geertsen. Fast Infoset. Technical report, June 2004. http://java.sun.com/developer/technicalArticles/xml/fastinfoset/. [cited at p. 28, 29, 41]

[70] Paul Sandoz, Santiago Pericas-Geertsen, Kohuske Kawaguchi, Marc Hadley, and Eduardo Pelegri-Llopart. Fast Web Services. Technical report, August 2003. http://java.sun.com/developer/technicalArticles/WebServices/fastWS/. [cited at p. 29]

[71] Sanjay Sarma, David L. Brock, and Kevin Ashton. The Networked Physical World, 2000. TR MIT-AUTOID-WH-001, MIT Auto-ID Center. [cited at p. 3]

[72] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002. [cited at p. 144]

[73] John Schneider, Takuki Kamiya, Daniel Peintner, and Rumen Kyusakov. Efficient XML Interchange (EXI) Format 1.0 (Second Edition). W3C Recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-exi-20140211/. [cited at p. 31, 35, 36, 39, 41, 42, 51, 59, 71, 80, 96, 100, 114, 119, 139, 154]

[74] Andreas Scholz, Christian Buckl, Irina Gaponova, Stephan Sommer, Alois Knoll, Alfons Kemper, Jörg Heuer, and Anton Schmitt. An Adaptive SOA for Embedded Networks. INDIN '09: 7th IEEE International Conference on Industrial Informatics, 2009. [cited at p. 8]

[75] Andreas Scholz, Stephan Sommer, Alfons Kemper, Alois Knoll, Christian Buckl, Gerd Kainz, Jörg Heuer, and Anton Schmitt. Towards an adaptive execution of applications in heterogeneous embedded networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 26–31. ACM, 2010. [cited at p. 8]

[76] C. M. Sperberg-McQueen, Henry S. Thompson, Murray Maloney, Henry S. Thompson, David Beech, Noah Mendelsohn, and Shudi (Sandy) Gao. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. Last call WD, W3C, December 2009. http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/. [cited at p. 17, 96]

[77] C. Timmerer, H. Hellwagner, J. Heuer, C. Seyrat, and A. Hutter. BinaryXML - A Comparison of Existing XML Compression Techniques. MPEG Contribution, Munich, Germany, 2004. ISO/IEC JTCI/SC29/WG11 MPEG2004/M10718. [cited at p. 29]

[78] Richard Tobin and John Cowan. XML Information Set. First Edition of a Recommendation, W3C, October 2001. http://www.w3.org/TR/2001/REC-xml-infoset-20011024/. [cited at p. 31, 140]

[79] Richard Tobin and John Cowan. XML Information Set (Second Edition). W3C Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204. [cited at p. 31, 71, 141]

[80] Daniel Veillard and Paul Grosso. XML Fragment Interchange. Candidate Recommendation, W3C, February 2001. http://www.w3.org/TR/2001/CR-xml-fragment-20010212. [cited at p. 29]

[81] Stephen Williams, Greg White, Don Brutzman, and Jaakko Kangasharju. Efficient XML Interchange Measurements Note. W3C Working Draft, W3C, July 2007. http://www.w3.org/TR/2007/WD-exi-measurements-20070725/. [cited at p. 20, 22, 23, 31, 61, 75, 76]

[82] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. [cited at p. 25]

List of Symbols and Abbreviations

Abbreviation	Description	Definition
ABNF	Augmented Backus-Naur Form	page 79
API	Application Programming Interface	page 53
ASN.1	Abstract Syntax Notation One	page 28
BEV	Battery Electric Vehicles	page 4
BiM	Binary MPEG format for XML	page 29
BMU	Bundesministerium für Umwelt, Naturschutz und Reaktorsicherheit	page 10
BMWi	Bundesministerium für Wirtschaft und Technologie	page 10
EBNF	Extended Backus-Naur Form	page 79
EVSE	Electric Vehicle Supply Equipment	page 4
BNF	Backus Normal Form or Backus-Naur Form	page 81
CLOB	Collection of character data	page 62
DIN	Deutsche Institut für Normung	page 157
DOM	Document Object Model	page 53
DVB	Digital Video Broadcasting	page 30
EXI	Efficient XML Interchange	page 30
HTML	HyperText Markup Language	page 72
ITU	International Telecommunication Union	page 28
ISO	International Organization for Standardization	page 28
JAR	Java ARchive, aggregates many files into one to distribute Java applications or libraries	page 60
JSON	JavaScript Object Notation	page 155
LZ77	Kind of Ziv–Lempl algorithms	page 25
MPEG	Moving Picture Experts Group	page 29
OMA	Open Mobile Alliance	page 28
PHEV	Plug-in Hybrid Electric Vehicles	page 4
RIA	Rich Internet Application	page 72
SAX	Simple API for XML	page 53

Abbreviation	Description	Definition
SOA	Service-oriented Architecture	page 8
SOAP	Simple Object Access Protocol	page 8
StAX	Streaming API for XML	page 53
STX	Streaming Transformations for XML	page 150
SyncML	Synchronization Markup Language	page 28
SVG	Scalable Vector Graphics	page 74
URI	Uniform Resource Identifier	page 92
VoIP	Voice over IP	page 154
WBXML	WAP Binary XML	page 28
WAP	Wireless Application Protocol	page 28
W3C	World Wide Web Consortium	page 15
XBC	XML Binary Characterization, W3C Working Group	page 30
XHTML	Extensible HyperText Markup Language	page 72
XMPP	Extensible Messaging and Presence Protocol	page 154
XPath	XML Path Language	page 139
XSLT	Extensible Stylesheet Language Transformations	page 139
XQuery	XQuery is a query and functional programming language that is designed to query collections of XML data	page 139

Appendices

Verbose Listings

The following notation is used in the Backus-Naur Form (BNF) description of the grammar for SVG path data[4].

Listing 1: BNF for SVG paths

```
svg-path:
    wsp* moveto-drawto-command-groups? wsp*
moveto-drawto-command-groups:
    moveto-drawto-command-group
    | moveto-drawto-command-group wsp* moveto-drawto-command-groups
moveto-drawto-command-group:
    moveto wsp* drawto-commands?
drawto-commands:
    drawto-command
    | drawto-command wsp* drawto-commands
drawto-command:
    closepath
    | lineto
    | horizontal-lineto
    | vertical-lineto
    | curveto
    | smooth-curveto
    | quadratic-bezier-curveto
    | smooth-quadratic-bezier-curveto
    | elliptical-arc
moveto:
    ( "M" | "m" ) wsp* moveto-argument-sequence
moveto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
closepath:
    ("Z" | "z")
lineto:
    ( "L" | "l" ) wsp* lineto-argument-sequence
lineto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
horizontal-lineto:
    ( "H" | "h" ) wsp* horizontal-lineto-argument-sequence
horizontal-lineto-argument-sequence:
```

[4] http://www.w3.org/TR/SVG2/paths.html#PathDataBNF

```
    coordinate
    | coordinate comma-wsp? horizontal-lineto-argument-sequence
vertical-lineto:
    ( "V" | "v" ) wsp* vertical-lineto-argument-sequence
vertical-lineto-argument-sequence:
    coordinate
    | coordinate comma-wsp? vertical-lineto-argument-sequence
curveto:
    ( "C" | "c" ) wsp* curveto-argument-sequence
curveto-argument-sequence:
    curveto-argument
    | curveto-argument comma-wsp? curveto-argument-sequence
curveto-argument:
    coordinate-pair comma-wsp? coordinate-pair comma-wsp? coordinate-pair
smooth-curveto:
    ( "S" | "s" ) wsp* smooth-curveto-argument-sequence
smooth-curveto-argument-sequence:
    smooth-curveto-argument
    | smooth-curveto-argument comma-wsp? smooth-curveto-argument-sequence
smooth-curveto-argument:
    coordinate-pair comma-wsp? coordinate-pair
quadratic-bezier-curveto:
    ( "Q" | "q" ) wsp* quadratic-bezier-curveto-argument-sequence
quadratic-bezier-curveto-argument-sequence:
    quadratic-bezier-curveto-argument
    | quadratic-bezier-curveto-argument comma-wsp?
        quadratic-bezier-curveto-argument-sequence
quadratic-bezier-curveto-argument:
    coordinate-pair comma-wsp? coordinate-pair
smooth-quadratic-bezier-curveto:
    ( "T" | "t" ) wsp* smooth-quadratic-bezier-curveto-argument-sequence
smooth-quadratic-bezier-curveto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? smooth-quadratic-bezier-curveto-argument-sequence
elliptical-arc:
    ( "A" | "a" ) wsp* elliptical-arc-argument-sequence
elliptical-arc-argument-sequence:
    elliptical-arc-argument
    | elliptical-arc-argument comma-wsp? elliptical-arc-argument-sequence
elliptical-arc-argument:
    nonnegative-number comma-wsp? nonnegative-number comma-wsp?
        number comma-wsp flag comma-wsp? flag comma-wsp? coordinate-pair
coordinate-pair:
    coordinate comma-wsp? coordinate
coordinate:
    number
nonnegative-number:
    integer-constant
    | floating-point-constant
number:
    sign? integer-constant
    | sign? floating-point-constant
flag:
    "0" | "1"
comma-wsp:
    (wsp+ comma? wsp*) | (comma wsp*)
comma:
    ","
integer-constant:
```

```
    digit-sequence
floating-point-constant:
    fractional-constant exponent?
    | digit-sequence exponent
fractional-constant:
    digit-sequence? "." digit-sequence
    | digit-sequence "."
exponent:
    ( "e" | "E" ) sign? digit-sequence
sign:
    "+" | "-"
digit-sequence:
    digit
    | digit digit-sequence
digit:
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
wsp:
    (#x20 | #x9 | #xD | #xA)
```

The following XML schema document is generated by the automatic datatype representation generator shown in Chapter 8 according to the BNF in Listing 8.2.

Listing 2: XML schema for xsd:date

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="xsdDate">
      <xs:complexType>
         <xs:sequence>
            <!-- year = [negative] , digit , {digit} -->
            <xs:element ref="year"/>
            <!-- '-' -->
            <!-- month = digit , {digit} -->
            <xs:element ref="month"/>
            <!-- '-' -->
            <!-- day = digit , {digit} -->
            <xs:element ref="day"/>
            <!-- [ timezone ] -->
            <xs:sequence minOccurs="0">
               <!-- timezone = 'Z' | ( '+' | '-' ) , hour , ':' , minute -->
               <xs:element ref="timezone"/>
            </xs:sequence>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="year" type="xs:short"/>
   <xs:element name="month" type="xs:unsignedByte" />
   <xs:element name="day" type="xs:unsignedByte" />
   <xs:element name="timezone">
      <xs:complexType>
         <xs:sequence>
            <!-- 'Z' | ( '+' | '-' ) , hour , ':' , minute -->
            <xs:choice>
               <!-- Anonymous = 'Z' -->
               <xs:element name="Anon_0">
                  <xs:complexType>
                     <xs:sequence>
                        <!-- 'Z' -->
                        <xs:element name="terminal0">
                           <xs:simpleType>
                              <xs:restriction base="xs:string">
                                 <xs:enumeration value="Z"/>
                              </xs:restriction>
                           </xs:simpleType>
                        </xs:element>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
               <!-- Anonymous = ( '+' | '-' ) , hour , ':' , minute -->
               <xs:element name="Anon_1">
                  <xs:complexType>
                     <xs:sequence>
                        <!-- '+' | '-' -->
                        <xs:sequence>
                           <!-- '+' | '-' -->
                           <xs:choice>
                              <!-- '+' -->
                              <xs:element name="terminal0">
                                 <xs:simpleType>
                                    <xs:restriction base="xs:string">
```

```
                                                <xs:enumeration value="+"/>
                                            </xs:restriction>
                                        </xs:simpleType>
                                    </xs:element>
                                </xs:choice>
                            </xs:sequence>
                            <!-- hour = digit , {digit} -->
                            <xs:element ref="hour"/>
                            <!-- ’:’ -->
                            <!-- minute = digit , {digit} -->
                            <xs:element ref="minute"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="hour" type="xs:unsignedByte"> </xs:element>
<xs:element name="minute" type="xs:unsignedByte"> </xs:element>
<xs:element name="negative">
    <xs:complexType>
        <xs:sequence>
            <!-- ’-’ -->
            <xs:element name="terminal0">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="-"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="sign">
    <xs:complexType>
        <xs:sequence>
            <!-- ’+’ | ’-’ -->
            <xs:choice>
                <!-- ’+’ -->
                <xs:element name="terminal0">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="+"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <!-- ’-’ -->
                <xs:element name="terminal1">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="-"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
</xs:schema>
```

The following XML schema document is also generated by the automatic datatype representation generator shown in Chapter 8.

Listing 3: XML schema for SVGPath d Attribute

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
<!--
  svgPath = [ movetoDrawtoCommandGroups ];
  movetoDrawtoCommandGroups = movetoDrawtoCommandGroup , { {wsp} , {
       movetoDrawtoCommandGroup} };
  movetoDrawtoCommandGroup = moveto , { {wsp} , { drawtoCommands } };
  drawtoCommands = drawtoCommand , { {wsp} , {drawtoCommand} };
  drawtoCommand = closepath | lineto | horizontalLineto | verticalLineto | curveto |
       smoothCurveto | quadraticBezierCurveto | smoothQuadraticBezierCurveto |
       ellipticalArc;
  moveto = ( 'M' | 'm' ) , {wsp} , coordinatePair, { {commaWsp} , {coordinatePair} };
  closepath = ('Z' | 'z');
  lineto = ( 'L' | 'l' ) , {wsp} , coordinatePair , { {commaWsp} , {coordinatePair} };
  horizontalLineto = ( 'H' | 'h' ) , {wsp} , coordinate , { {commaWsp} , {coordinate} };
  verticalLineto = ( 'V' | 'v' ) , {wsp} , coordinate , { {commaWsp} , {coordinate} };
  curveto = ( 'C' | 'c' ) , {wsp} , coordinatePair , { {commaWsp} , {coordinatePair} };
  smoothCurveto = ( 'S' | 's' ) , {wsp} , coordinatePair , { {commaWsp} , {coordinatePair}
        };
  quadraticBezierCurveto = ( 'Q' | 'q' ) , {wsp} , coordinatePair , { {commaWsp} , {
       coordinatePair} };
  smoothQuadraticBezierCurveto = ( 'T' | 't' ) , {wsp} , coordinatePair , { {commaWsp} , {
       coordinatePair} };
  ellipticalArc = ( 'A' | 'a' ) , {wsp} , ellipticalArcArgument , { {commaWsp} , {
       ellipticalArcArgument} };
  ellipticalArcArgument = nonnegativeNumber , { commaWsp } , nonnegativeNumber , {
       commaWsp } , number , { commaWsp } , flag , { commaWsp } , flag , { commaWsp } ,
       coordinatePair;
  coordinatePair = coordinate , { commaWsp } , coordinate;
  coordinate = number;
  nonnegativeNumber = digitSequence , { '.' , [ digitSequence ] } , { [ exponent ] |
       digitSequence , exponent };
  number = [sign] , digitSequence , { ( '.' | exponent ) , digitSequence };
  flag = '0' | '1';
  commaWsp = ',' | ' ' | ' ' | '
' | '
';
  comma = ',';
  integerConstant = digitSequence;
  floatingPointConstant = fractionalConstant , [ exponent ] | digitSequence , exponent;
  fractionalConstant = [ digitSequence ] , '.' , [ digitSequence ];
  exponent = ( 'e' | 'E' ), [ sign ] , digitSequence;
  sign = '+' | '-';
  digitSequence = digit , { digit };
  digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
  wsp = ' ' | ' ' | '
' | '
';
 -->
<xs:element name='svgPath' >
  <xs:complexType>
    <xs:sequence>
      <!-- [ movetoDrawtoCommandGroups ] -->
      <xs:sequence minOccurs='0'>
      <!-- movetoDrawtoCommandGroups = movetoDrawtoCommandGroup , { {wsp} , {
           movetoDrawtoCommandGroup} } -->
```

```
      <xs:element ref='movetoDrawtoCommandGroups' />
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='movetoDrawtoCommandGroups' >
  <xs:complexType>
    <xs:sequence>
      <!-- movetoDrawtoCommandGroup = moveto , { {wsp} , { drawtoCommands } } -->
      <xs:element ref='movetoDrawtoCommandGroup' />
      <!-- { {wsp} , {movetoDrawtoCommandGroup} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { movetoDrawtoCommandGroup } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- movetoDrawtoCommandGroup = moveto , { {wsp} , { drawtoCommands } } -->
      <xs:element ref='movetoDrawtoCommandGroup' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='movetoDrawtoCommandGroup' >
  <xs:complexType>
    <xs:sequence>
      <!-- moveto = ( 'M' | 'm' ) , {wsp} , coordinatePair, { {commaWsp} , {coordinatePair
           } } -->
      <xs:element ref='moveto' />
      <!-- { {wsp} , { drawtoCommands } } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { drawtoCommands } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- drawtoCommands = drawtoCommand , { {wsp} , {drawtoCommand} } -->
      <xs:element ref='drawtoCommands' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='drawtoCommands' >
  <xs:complexType>
    <xs:sequence>
      <!-- drawtoCommand = closepath | lineto | horizontalLineto | verticalLineto |
           curveto | smoothCurveto | quadraticBezierCurveto | smoothQuadraticBezierCurveto
           | ellipticalArc -->
      <xs:element ref='drawtoCommand' />
      <!-- { {wsp} , {drawtoCommand} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
```

```
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { drawtoCommand } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- drawtoCommand = closepath | lineto | horizontalLineto | verticalLineto |
           curveto | smoothCurveto | quadraticBezierCurveto | smoothQuadraticBezierCurveto
           | ellipticalArc -->
      <xs:element ref='drawtoCommand' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='drawtoCommand' >
  <xs:complexType>
    <xs:sequence>
      <!-- closepath | lineto | horizontalLineto | verticalLineto | curveto |
           smoothCurveto | quadraticBezierCurveto | smoothQuadraticBezierCurveto |
           ellipticalArc -->
      <xs:choice>
      <!-- closepath = ('Z' | 'z') -->
      <xs:element ref='closepath' />
      <!-- lineto = ( 'L' | 'l' ) , {wsp} , coordinatePair , { {commaWsp} , {
           coordinatePair} } -->
      <xs:element ref='lineto' />
      <!-- horizontalLineto = ( 'H' | 'h' ) , {wsp} , coordinate , { {commaWsp} , {
           coordinate} } -->
      <xs:element ref='horizontalLineto' />
      <!-- verticalLineto = ( 'V' | 'v' ) , {wsp} , coordinate , { {commaWsp} , {
           coordinate} } -->
      <xs:element ref='verticalLineto' />
      <!-- curveto = ( 'C' | 'c' ) , {wsp} , coordinatePair , { {commaWsp} , {
           coordinatePair} } -->
      <xs:element ref='curveto' />
      <!-- smoothCurveto = ( 'S' | 's' ) , {wsp} , coordinatePair , { {commaWsp} , {
           coordinatePair} } -->
      <xs:element ref='smoothCurveto' />
      <!-- quadraticBezierCurveto = ( 'Q' | 'q' ) , {wsp} , coordinatePair , { {commaWsp}
           , {coordinatePair} } -->
      <xs:element ref='quadraticBezierCurveto' />
      <!-- smoothQuadraticBezierCurveto = ( 'T' | 't' ) , {wsp} , coordinatePair , { {
           commaWsp} , {coordinatePair} } -->
      <xs:element ref='smoothQuadraticBezierCurveto' />
      <!-- ellipticalArc = ( 'A' | 'a' ) , {wsp} , ellipticalArcArgument , { {commaWsp} ,
           {ellipticalArcArgument} } -->
      <xs:element ref='ellipticalArc' />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='moveto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'M' | 'm' -->
      <xs:sequence>
```

```
      <!-- 'M' | 'm' -->
      <xs:choice>
      <!-- 'M' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='M'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'm' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='m'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='closepath' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'Z' | 'z' -->
      <xs:sequence>
      <!-- 'Z' | 'z' -->
      <xs:choice>
      <!-- 'Z' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='Z'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
```

```
      </xs:element>
      <!-- 'z' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='z'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='lineto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'L' | 'l' -->
      <xs:sequence>
      <!-- 'L' | 'l' -->
      <xs:choice>
      <!-- 'L' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='L'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'l' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='l'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
```

```
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='horizontalLineto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'H' | 'h' -->
      <xs:sequence>
      <!-- 'H' | 'h' -->
      <xs:choice>
      <!-- 'H' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='H'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'h' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='h'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinate = number -->
      <xs:element ref='coordinate' />
      <!-- { {commaWsp} , {coordinate} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinate } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinate = number -->
      <xs:element ref='coordinate' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='verticalLineto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'V' | 'v' -->
```

```
      <xs:sequence>
      <!-- 'V' | 'v' -->
      <xs:choice>
      <!-- 'V' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='V'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'v' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='v'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinate = number -->
      <xs:element ref='coordinate' />
      <!-- { {commaWsp} , {coordinate} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinate } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinate = number -->
      <xs:element ref='coordinate' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='curveto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'C' | 'c' -->
      <xs:sequence>
      <!-- 'C' | 'c' -->
      <xs:choice>
      <!-- 'C' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='C'></xs:enumeration>
          </xs:restriction>
```

```
        </xs:simpleType>
      </xs:element>
      <!-- 'c' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='c'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='smoothCurveto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'S' | 's' -->
      <xs:sequence>
      <!-- 'S' | 's' -->
      <xs:choice>
      <!-- 'S' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='S'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 's' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='s'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
```

```
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='quadraticBezierCurveto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'Q' | 'q' -->
      <xs:sequence>
      <!-- 'Q' | 'q' -->
      <xs:choice>
      <!-- 'Q' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='Q'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'q' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='q'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
```

```
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='smoothQuadraticBezierCurveto' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'T' | 't' -->
      <xs:sequence>
      <!-- 'T' | 't' -->
      <xs:choice>
      <!-- 'T' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='T'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 't' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='t'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      <!-- { {commaWsp} , {coordinatePair} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
```

```
      </xs:sequence>
      <!-- { coordinatePair } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='ellipticalArc' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'A' | 'a' -->
      <xs:sequence>
      <!-- 'A' | 'a' -->
      <xs:choice>
      <!-- 'A' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='A'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- 'a' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='a'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- { wsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- wsp = ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- ellipticalArcArgument = nonnegativeNumber , { commaWsp } , nonnegativeNumber ,
           { commaWsp } , number , { commaWsp } , flag , { commaWsp } , flag , { commaWsp }
            , coordinatePair -->
      <xs:element ref='ellipticalArcArgument' />
      <!-- { {commaWsp} , {ellipticalArcArgument} } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- { ellipticalArcArgument } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- ellipticalArcArgument = nonnegativeNumber , { commaWsp } , nonnegativeNumber ,
           { commaWsp } , number , { commaWsp } , flag , { commaWsp } , flag , { commaWsp }
            , coordinatePair -->
      <xs:element ref='ellipticalArcArgument' />
```

```
      </xs:sequence>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='ellipticalArcArgument' >
  <xs:complexType>
    <xs:sequence>
      <!-- nonnegativeNumber = digitSequence , { '.' , [ digitSequence ] } , { [ exponent
          ] | digitSequence , exponent } -->
      <xs:element ref='nonnegativeNumber' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- nonnegativeNumber = digitSequence , { '.' , [ digitSequence ] } , { [ exponent
          ] | digitSequence , exponent } -->
      <xs:element ref='nonnegativeNumber' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- number = [sign] , digitSequence , { ( '.' | exponent ) , digitSequence } -->
      <xs:element ref='number' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- flag = '0' | '1' -->
      <xs:element ref='flag' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- flag = '0' | '1' -->
      <xs:element ref='flag' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinatePair = coordinate , { commaWsp } , coordinate -->
      <xs:element ref='coordinatePair' />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='coordinatePair' >
  <xs:complexType>
    <xs:sequence>
      <!-- coordinate = number -->
```

```
      <xs:element ref='coordinate' />
      <!-- { commaWsp } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- commaWsp = ',' | ' ' | ' ' | '
' | '
' -->
      </xs:sequence>
      <!-- coordinate = number -->
      <xs:element ref='coordinate' />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='coordinate' type='xs:float' >
</xs:element>
<xs:element name='nonnegativeNumber' type='xs:float' >
</xs:element>
<xs:element name='number' >
  <xs:complexType>
    <xs:sequence>
      <!-- [ sign ] -->
      <xs:sequence minOccurs='0'>
      <!-- sign = '+' | '-' -->
      <xs:element ref='sign' />
      </xs:sequence>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
      <!-- { ( '.' | exponent ) , digitSequence } -->
      <xs:sequence minOccurs='0' maxOccurs='unbounded' >
      <!-- '.' | exponent -->
      <xs:sequence>
      <!-- '.' | exponent -->
      <xs:choice>
      <!-- Anonymous = '.' -->
      <xs:element name='Anon_0' >
        <xs:complexType>
          <xs:sequence>
      <!-- '.' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='.'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <!-- Anonymous = exponent -->
      <xs:element name='Anon_1' >
        <xs:complexType>
          <xs:sequence>
      <!-- exponent = ( 'e' | 'E' ), [ sign ] , digitSequence -->
      <xs:element ref='exponent' />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- digitSequence = digit , { digit } -->
```

```
      <xs:element ref='digitSequence' />
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='flag' >
  <xs:complexType>
    <xs:sequence>
      <!-- '0' | '1' -->
      <xs:choice>
      <!-- '0' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='0'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- '1' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='1'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='commaWsp' >
  <xs:complexType>
    <xs:sequence>
      <!-- ',' | ' ' | ' ' | '
' | '
' -->
      <xs:choice>
      <!-- ',' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value=','></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- ' ' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='&#032;'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- ' ' -->
      <xs:element name='terminal2'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='&#009;'></xs:enumeration>
          </xs:restriction>
```

```
        </xs:simpleType>
      </xs:element>
      <!-- '
' -->
      <xs:element name='terminal3'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='&#x0D;'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- '
' -->
      <xs:element name='terminal4'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='&#x0A;'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='comma' >
  <xs:complexType>
    <xs:sequence>
      <!-- ',' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value=','></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='integerConstant' >
  <xs:complexType>
    <xs:sequence>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='floatingPointConstant' >
  <xs:complexType>
    <xs:sequence>
      <!-- fractionalConstant , [ exponent ] | digitSequence , exponent -->
      <xs:choice>
      <!-- Anonymous = fractionalConstant , [ exponent ] -->
      <xs:element name='Anon_0' >
        <xs:complexType>
          <xs:sequence>
      <!-- fractionalConstant = [ digitSequence ] , '.' , [ digitSequence ] -->
      <xs:element ref='fractionalConstant' />
      <!-- [ exponent ] -->
      <xs:sequence minOccurs='0'>
```

```
      <!-- exponent = ( 'e' | 'E' ), [ sign ] , digitSequence -->
      <xs:element ref='exponent' />
      </xs:sequence>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <!-- Anonymous = digitSequence , exponent -->
      <xs:element name='Anon_1' >
        <xs:complexType>
          <xs:sequence>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
      <!-- exponent = ( 'e' | 'E' ), [ sign ] , digitSequence -->
      <xs:element ref='exponent' />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='fractionalConstant' >
  <xs:complexType>
    <xs:sequence>
      <!-- [ digitSequence ] -->
      <xs:sequence minOccurs='0'>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
      </xs:sequence>
      <!-- '.' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='.'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- [ digitSequence ] -->
      <xs:sequence minOccurs='0'>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='exponent' >
  <xs:complexType>
    <xs:sequence>
      <!-- 'e' | 'E' -->
      <xs:sequence>
      <!-- 'e' | 'E' -->
      <xs:choice>
      <!-- 'e' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='e'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
```

```
      </xs:element>
      <!-- 'E' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='E'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
      </xs:sequence>
      <!-- [ sign ] -->
      <xs:sequence minOccurs='0'>
      <!-- sign = '+' | '-' -->
      <xs:element ref='sign' />
      </xs:sequence>
      <!-- digitSequence = digit , { digit } -->
      <xs:element ref='digitSequence' />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='sign' >
  <xs:complexType>
    <xs:sequence>
      <!-- '+' | '-' -->
      <xs:choice>
      <!-- '+' -->
      <xs:element name='terminal0'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='+'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <!-- '-' -->
      <xs:element name='terminal1'>
        <xs:simpleType>
          <xs:restriction base='xs:string'>
            <xs:enumeration value='-'></xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name='digitSequence' type='xs:integer' >
</xs:element>

</xs:schema>
```

Listing 4: SchemaForGrammars.xsd

```
<xs:schema targetNamespace="http://www.ct.siemens.com/2012/SchemaForGrammars"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:exi="http://www.ct.siemens.com/2012/SchemaForGrammars" elementFormDefault="
       qualified">

   <!-- ********************************************** -->
   <!-- Datatype -->
   <!-- ********************************************** -->
   <xs:complexType name="Datatype">
      <xs:choice>
         <xs:element name="datatypeBuiltInType" type="exi:builtInType"/>
         <xs:element name="datatypeNBitUnsignedInteger" type="exi:
             DatatypeNBitUnsignedInteger"/>
         <xs:element name="datatypeDateTime" type="exi:dateTimeType"/>
         <xs:element name="datatypeEnumeration" type="exi:DatatypeEnumeration"/>
         <xs:element name="datatypeList" type="exi:Datatype"/>
         <xs:element name="datatypeRestrictedCharSet" type="exi:
             DatatypeRestrictedCharSet"/>
      </xs:choice>
   </xs:complexType>

   <!-- ********************************************** -->
   <!-- EXI built-in datatypes -->
   <!-- ********************************************** -->
   <xs:simpleType name="builtInType">
      <xs:restriction base="xs:string">
         <!-- Binary -->
         <xs:enumeration value="BINARY_BASE64"/>
         <xs:enumeration value="BINARY_HEX"/>
         <!-- Boolean -->
         <xs:enumeration value="BOOLEAN"/>
         <xs:enumeration value="BOOLEAN_PATTERN"/>
         <!-- Decimal -->
         <xs:enumeration value="DECIMAL"/>
         <!-- Float -->
         <xs:enumeration value="FLOAT"/>
         <!-- NBit-Unsigned Integer -->
         <xs:enumeration value="NBIT_UNSIGNED_INTEGER"/>
         <!-- Unsigned Integer -->
         <xs:enumeration value="UNSIGNED_INTEGER"/>
         <!-- (Signed) Integer -->
         <xs:enumeration value="INTEGER"/>
         <!-- Datetime -->
         <xs:enumeration value="DATETIME"/>
         <!-- String -->
         <xs:enumeration value="STRING"/>
         <!-- Enumeration -->
         <xs:enumeration value="ENUMERATION"/>
         <!-- List -->
         <xs:enumeration value="LIST"/>
         <!-- Restricted Character Set -->
         <xs:enumeration value="RESTRICTED_CHARACTER_SET"/>
         <!-- QName -->
         <xs:enumeration value="QNAME"/>
      </xs:restriction>
   </xs:simpleType>

   <!-- ********************************************** -->
```

```
<!-- N-Bit Unsigned Integer datatype -->
<!-- ********************************************** -->
<xs:complexType name="DatatypeNBitUnsignedInteger">
   <xs:sequence>
      <xs:element name="lowerBound" type="xs:integer"/>
      <xs:element name="upperBound" type="xs:integer"/>
   </xs:sequence>
</xs:complexType>

<!-- ********************************************** -->
<!-- Date-Time datatype -->
<!-- ********************************************** -->
<xs:simpleType name="dateTimeType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="gYear"/>
      <xs:enumeration value="gYearMonth"/>
      <xs:enumeration value="date"/>
      <xs:enumeration value="dateTime"/>
      <xs:enumeration value="gMonth"/>
      <xs:enumeration value="gMonthDay"/>
      <xs:enumeration value="gDay"/>
      <xs:enumeration value="time"/>
   </xs:restriction>
</xs:simpleType>

<!-- ********************************************** -->
<!-- Enumeration datatype -->
<!-- ********************************************** -->
<xs:complexType name="DatatypeEnumeration">
   <xs:sequence>
      <xs:element name="enumValuesBuiltInType" type="exi:builtInType"/>
      <xs:element name="enumValues">
         <xs:simpleType>
            <xs:list itemType="xs:string"/>
         </xs:simpleType>
      </xs:element>
   </xs:sequence>
</xs:complexType>

<!-- ********************************************** -->
<!-- Restricted Character Set datatype -->
<!-- ********************************************** -->
<xs:complexType name="DatatypeRestrictedCharSet">
   <xs:sequence>
      <xs:element name="codePoint" type="xs:unsignedInt" minOccurs="0" maxOccurs="
          unbounded"/>
   </xs:sequence>
</xs:complexType>



<!-- ********************************************** -->
<!-- GRAMMAR PRODUCTION -->
<!-- ********************************************** -->
<xs:complexType name="Production">
   <xs:sequence>
      <!-- ********************************************** -->
      <!-- EXI Events -->
      <!-- ********************************************** -->
```

```
        <xs:choice>
            <xs:element name="startDocument"/>
            <xs:element name="endDocument"/>
            <xs:element name="startElement">
                <xs:complexType>
                    <xs:sequence>
                        <!-- element qname -->
                        <xs:element name="startElementQNameID" type="xs:unsignedInt"/>
                        <!-- element rule -->
                        <xs:element name="startElementGrammarID" type="xs:unsignedInt"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="startElementGeneric"/>
            <xs:element name="endElement"/>
            <xs:element name="characters">
                <xs:complexType>
                    <xs:sequence>
                        <!-- schema value-type qname -->
                        <xs:element name="charactersSchemaTypeQNameID" type="xs:
                             unsignedInt"/>
                        <!-- datatype -->
                        <xs:element name="charactersDatatype" type="exi:Datatype"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="charactersGeneric"/>
            <xs:element name="attribute">
                <xs:complexType>
                    <xs:sequence>
                        <!-- attribute qname -->
                        <xs:element name="attributeQNameID" type="xs:unsignedInt"/>
                        <!-- attribute schema value-type qname -->
                        <xs:element name="attributeSchemaTypeQNameID" type="xs:
                             unsignedInt"/>
                        <!-- datatype -->
                        <xs:element name="attributeDatatype" type="exi:Datatype"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="attributeGeneric"/>
        </xs:choice>
        <!-- No nextRule for ED and EE -->
        <xs:element name="nextGrammarID" type="xs:unsignedInt" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<!-- startTag grammars and element grammars -->
<xs:complexType name="Grammar">
    <xs:sequence>
        <xs:element name="production" type="exi:Production" minOccurs="0" maxOccurs="
             unbounded"
        />
    </xs:sequence>
</xs:complexType>

<!-- first startTag grammar -->
<xs:complexType name="FirstStartTagGrammar">
    <xs:complexContent>
```

```
        <xs:extension base="exi:Grammar">
            <xs:attribute name="isTypeCastable" type="xs:boolean" use="required"/>
            <xs:attribute name="isNillable" type="xs:boolean" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>


<!-- ********************************************** -->
<!-- EXI Grammar (root element) -->
<!-- ********************************************** -->
<xs:element name="exiGrammar">
    <xs:complexType>
        <xs:sequence>
            <!-- ********************************************** -->
            <!-- QNAMES -->
            <!-- ********************************************** -->
            <xs:element name="qnames">
                <xs:complexType>
                    <xs:sequence>
                        <!-- NamespaceUri's, prefixes and localNames (QName) -->
                        <xs:element name="namespaceContext" minOccurs="0" maxOccurs="
                             unbounded">
                            <xs:complexType>
                                <xs:sequence>
                                    <!-- namespace URI-->
                                    <xs:any processContents="skip"/>
                                    <!-- prefix -->
                                    <xs:element name="prefix" type="xs:string" minOccurs="
                                         0"/>
                                    <!-- QName Context -->
                                    <xs:element name="qnameContext" minOccurs="0"
                                        maxOccurs="unbounded">
                                        <xs:complexType>
                                            <xs:sequence>
                                              <!-- local-name -->
                                              <xs:any processContents="skip"/>
                                              <!-- global type (if any) -->
                                              <xs:element name="globalTypeGrammarID"
                                              type="xs:unsignedInt" minOccurs="0"/>
                                              <!-- global element (if any) -->
                                              <xs:element name="globalElementGrammarID"
                                              type="xs:unsignedInt" minOccurs="0"/>
                                              <!-- global attribute if any -->
                                              <xs:sequence minOccurs="0">
                                              <xs:element
                                              name="globalAttributeSchemaTypeQNameID"
                                              type="xs:unsignedInt"/>
                                              <xs:element name="globalAttributeDatatype"
                                              type="exi:Datatype"/>
                                              </xs:sequence>
                                              <!-- simple sub-types -->
                                              <xs:element name="simpleSubTypeQNameIDs"
                                              minOccurs="0">
                                              <xs:simpleType>
                                              <xs:list itemType="xs:unsignedInt"/>
                                              </xs:simpleType>
                                              </xs:element>
                                            </xs:sequence>
```

```
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name="numberOfLocalNames" type="xs:
                             unsignedInt"
                            use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <!-- number of entries -->
            <xs:attribute name="numberOfUris" type="xs:unsignedInt" use="
                 required"/>
            <xs:attribute name="numberOfQNames" type="xs:unsignedInt" use="
                 required"/>
        </xs:complexType>
    </xs:element>

    <!-- ********************************************** -->
    <!-- GRAMMAR RULES -->
    <!-- ********************************************** -->
    <xs:element name="grammars">
        <xs:complexType>
            <xs:sequence>
                <!-- document -->
                <xs:element name="document" type="exi:Grammar"/>
                <xs:element name="docContent" type="exi:Grammar"/>
                <xs:element name="docEnd" type="exi:Grammar"/>
                <!-- fragment -->
                <xs:element name="fragment" type="exi:Grammar"/>
                <xs:element name="fragmentContent" type="exi:Grammar"/>
                <!-- first startTag content -->
                <xs:element name="firstStartTagContent" type="exi:
                     FirstStartTagGrammar"
                    minOccurs="0" maxOccurs="unbounded"/>
                <!-- startTag content -->
                <xs:element name="startTagContent" type="exi:Grammar" minOccurs="
                     0"
                    maxOccurs="unbounded"/>
                <!-- element content -->
                <xs:element name="elementContent" type="exi:Grammar" minOccurs="0
                     "
                    maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="numberOfGrammars" type="xs:unsignedInt" use="
                 required"/>
        </xs:complexType>
    </xs:element>

  </xs:sequence>
 </xs:complexType>
</xs:element>


</xs:schema>
```