BSSC 2004(3) Issue 10

Design & Style Guide for XML Data and Schema

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

Document Status Sheet

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC 2004(3) Issue 10			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
0	1	23/10/2004	New document
0	2	08/12/2004	Initial formatting.
0	3	04/03/2005	Structure and rule content clean-up.
0	4	11/03/2005	General document clean-up, content and formatting harmonizing and style correction.
0	5	31/03/2005	Some inconsistencies fixed. Introductory chapters restructured into a single one. Additional restructuring to group rules that cover the same topic. Stylistic corrections.

Approved, January 13rd 2005
Board for Software Standardisation and Control
Ms M. Spada, BSSC chairman

Copyright © 2005 by European Space Agency

Table of Contents

DOCUMENT STATUS SHEET II

TABLE OF CONTENTS III

LIST OF RULES VI

LIST OF RECOMMENDATIONS VIII

PREFACE X

CHAPTER 1
INTRODUCTION 11

1.1 SCOPE AND APPLICABILITY 11
1.2 POSITION OF THIS DOCUMENT WITH RESPECT TO THE ECSS-E40 11
1.3 DOCUMENT OVERVIEW 12
1.4 GLOSSARY 12
1.5 ACRONYMS 13

CHAPTER 2
GENERAL PROJECT GUIDELINES 15

2.1 INTRODUCTION 15
2.2 PROJECT RULES AND RECOMMENDATIONS 15

CHAPTER 3
INTRODUCTION TO THE XML STANDARDS 17

3.1 BENEFITS OF THE USE OF XML 17
3.2 DRAWBACKS OF THE USE OF XML 18
3.3 USE OF XML WITHIN ESA SPACE PROJECTS 19
3.4 OVERVIEW OF THE XML TECHNOLOGIES 19
3.5 BINARY ENCODING OF XML 21
3.6 XML SYNTAX RULES 21
3.6.1 Well-formed XML 22
3.6.2 Processing Instructions 22
3.6.3 Comments 22

CHAPTER 4
XML TOOLS AND IDES 23

4.1 CREATION AND MODIFICATION: XML EDITORS 23
4.2 SYNTAX CHECKERS: XML PARSERS 23
4.3 SEMANTIC CHECKERS: XML VALIDATORS 23

4.4 XML BROWSERS 23
4.5 PUBLISHING SOFTWARE 23
4.6 ARCHIVING, STORAGE AND RETRIEVAL SOFTWARE 23
4.7 SECURITY TOOLS 23
4.8 SEARCHING AND CRAWLING SOFTWARE 24
4.9 DISTRIBUTION SOFTWARE 24
4.10 ROUTING AND AUDITING SOFTWARE 24
4.11 DOCUMENTATION SOFTWARE 24
4.12 GRAPHICAL IDES 24

CHAPTER 5
NAMING 25

5.1 GENERAL NAMING CONVENTIONS 25
5.2 XML DATA 26
5.2.1 Element Names 26
5.2.2 Attribute Names 27
5.3 NAMESPACES 27

CHAPTER 6
XML AND XML SCHEMA DESIGN AND STYLE 28

6.1 XML TREE DESIGN 28
6.2 ATTRIBUTES AND CHILD ELEMENTS 29
6.3 XML TYPES 30
6.4 NAMESPACE AND NAMESPACE REFERRALS 35
6.5 INTEGRATING XML SCHEMAS 37
6.6 EXTENSIBLE CONTENT MODELS FOR XML SCHEMA'S 37
6.7 EXPRESSING ADDITIONAL CONSTRAINTS 37
6.7.1 Supplement with Another Schema Language 38
6.7.2 Write Code to Express Additional Constraints 38
6.7.3 Express Additional Constraints with an XSLT/XPath Stylesheet 39
6.8 SCHEMA VERSIONING 39
6.8.1 Schema Versioning Techniques 41
6.8.2 Change the (Internal) Schema Version Attribute 42
6.8.3 Create a schemaVersion Attribute on the Root Element 42
6.8.4 Change the schema's targetNamespace. 44
6.8.5 Change the Name/Location of the Schema. 44
6.9 GLOBAL VERSUS LOCAL DEFINITIONS 45

CHAPTER 7
SECURITY-ENCRYPTIONS-KEYS 50

CHAPTER 8
XML AND BINARY DATA 51

8.1 INTRODUCTION 51
8.2 BINARY ATTACHMENTS 52
8.3 XML DATA COMPRESSION 52

8.4 OTHER BINARY ENCODING APPROACHES 53

BIBLIOGRAPHY... 54

List of Rules

Rule 1: Use meaningful, familiar and consistent names. 25

Rule 2: Avoid making abbreviations for names, removing vowels. 25

Rule 3: In mixed-case names, capitalize the first letter of standard acronyms. 25

Rule 4: Avoid names that differ only in case. 26

Rule 5: Avoid using similar names, which may be easily confused or mistyped. 26

Rule 6: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in an element name. 26

Rule 7: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in an attribute name. 27

Rule 8: Make two identical copies of all your schemas, where the copies differ only in the value of elementFormDefault (in one copy set elementFormDefault="qualified", in the other copy set elementFormDefault="unqualified"). 28

Rule 9: Uniquely identify all schema components with the id attribute. 29

Rule 10: If the item is not intended to be an element in instance documents, then define it as a type. 30

Rule 11: If the item´s content is to be reused by other items then define it as a type. 30

Rule 12: If the item is intended to be used as an element in instance documents, and it is required that sometimes it be nillable and other times not, then it must be defined as a type. ... 31

Rule 13: If the item is intended to be used as an element in instance documents and other elements are to be allowed to substitute for the element, then it must be declared as an element. 31

Rule 14: Do not use notations 34

Rule 15: Do favor key/keyref/unique over ID/IDREF for identity constraints. 34

Rule 16: Do not use default or fixed values especially for types of xs:QName. 35

Rule 17: If you have a targetNamespace, make it the default namespace. 36

Rule 18: Do not hard code the identity of an imported schema. 37

Rule 19: Capture the schema version somewhere in the XML schema. 40

Rule 20: Include information in the instance data files, that makes it possible to determine which version (or versions) of the schema they are compatible with. 40

Rule 21: Make older versions of your XML schema available. 40

Rule 22: In situations where a new version of a schema makes backwards incompatible changes (e.g., a construct that was valid and meaningful for the previous schema does not validate against the new schema), make sure that older instances will not be accidentally validated against the new version. 40

Rule 23: Where minimizing size and coupling of components is of utmost concern then use the so-called Russian Doll design. 45

Rule 24: Where your task requires that you make available to instance document authors the option to use element substitution, then use the so-called Salami Slice design. 46

Rule 25: The so-called Venetian Blind design is the one to choose when your schemas require the flexibility to turn namespace exposure on or off with a simple switch, and where component reuse is important. 47

List of Recommendations

Recommendation 1: Document any deviations. Never break a rule without documenting it. ... 15

Recommendation 2: Use independent tools to provide additional warnings and information about the code. 15

Recommendation 3: Use appropriate verification tools to ensure that the code conforms to the rules and to catch potential problems as early as possible. 16

Recommendation 4: Make sure that any code that you use for debugging purposes does not have side effects. 16

Recommendation 5: Avoid long (e.g. more than 20 characters) names. 25

Recommendation 6: Names containing abbreviations should be considered carefully to avoid ambiguity. 26

Recommendation 7: Use nouns to name elements. 27

Recommendation 8: When using a standard namespace use the namespace prefix recommended by the corresponding schema author. Conversely, avoid using common prefixes when you are not referring to the namespace usually associated to them. 27

Recommendation 9: XML trees should have the right level of depth and of width. 28

Recommendation 10: Minimize the use of global elements and attributes. 29

Recommendation 11: Postpone decisions as long as possible: Postpone binding a type reference to an implementation, i.e., use dangling types. 30

Recommendation 12: When in doubt, make it a type. You can always create an element from the type, if needed. 30

Recommendation 13: Do use attribute groups and model groups. 32

Recommendation 14: Do use complex types and attribute declarations. 32

Recommendation 15: Use design by composition rather than subclassing. 33

Recommendation 16: If the decision is made to use subclassing, then avoid using derive-by-restriction. 33

Recommendation 17: If the decision is made to use subclassing, then limit the type hierarchy to maximum 3 levels. 34

Recommendation 18: Do carefully use substitution groups. 34

Recommendation 19: Do use elementFormDefault set to qualified and attributeFormDefault set to unqualified. 35

Recommendation 20: Do use XML namespaces as much as possible. Learn the correct way to use them. 35

Recommendation 21: If you intend your schema type definitions to be reused in a variety of contexts, do not give them a targetNamespace. This is the so-called chameleon design pattern. 35

Recommendation 22: Create extensible schemas. Do use wildcards to provide well defined points of extensibility. Use the <any> element. 37

Recommendation 23: Take into account that XML schemas will not be able to express all of your business rules. 38

Recommendation 24: When an XML schema is only extended, (e.g., new elements, attributes, extensions to an enumerated list, etc.) one should strive to not invalidate existing instance documents. 41

Recommendation 25: Adopt a convention for schema version identification to indicate whether the schema changed significantly (changes were not backwards compatible) or was only extended (changes were backwards compatible). 41

Recommendation 26: Consider your requirements carefully before deciding to look for a binary XML solution. In many cases, the standard text based XML encoding is adequate. . . 51

Recommendation 27: For embedding small amounts of binary data in an XML file, consider using a binary to text encoding land putting it directly in the XML stream. 52

Recommendation 28: For packaging large amounts of binary data together with textual XML data in an efficient way, consider using the XML-binary Optimized Packaging (XOP). …… 52

Recommendation 29: Consider using a standard compression algorithm to compress XML data before storing or transmitting it. 52

Preface

This Coding Standard is based upon the experience of applying XML and related technologies to the development of custom space software systems. Published experience and Industry best practice rules, as well as experience from in-house developments, were all taken into account to create this document.

The BSSC wishes to thank the European Space Research and Technology Centre (ESTEC), Noordwijk, The Netherlands, and in particular Peter Claes, for preparing the standard. The BSSC also thanks all those who contributed ideas for this standard. The BSSC members that have reviewed the standard: Mariella Spada, Michael Jones, Jean-Loup Terraillon, Jean Pierre Guignard, Jerome Dumas, Daniel Ponz, Daniel de Pablo and Lothar Winzer. The BSSC also wishes to thank the following ESA reviewers of this standard: <reviewer names go here> and the expert reviewer, editor, Martín Soto, from *Fraunhofer Institute for Experimental Software Engineering* (IESE).

Requests for clarifications, change proposals or any other comments concerning this standard should be addressed to:

BSSC/ESOC Secretariat
Attention of Ms M. Spada
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr J.-L. Terraillon
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

Chapter 1
Introduction

1.1 Scope and applicability

This standard presents rules and recommendations about the usage of XML and related technologies. Although many books and a variety of other documents describe the usage of such technologies in detail, they often concentrate on what is possible and not necessarily on what is desirable or acceptable. Particularly, appropriate guidelines that apply to large software engineering projects intended for mission- or safety-critical systems, can be hard to find.

This document provides a set of guidelines for the design and implementation of XML data models, which are intended to improve the overall quality and maintainability of software systems developed by, or under contract to, the European Space Agency. The use of this standard should improve consistency across different software systems, potentially developed by different programming teams in different companies.

The guidelines in this standard should be met for XML related code and specifications to fully comply with this standard. The standard has no contractual implication. Contractual obligations are given in individual project documents.

1.2 Position of this document with respect to the ECSS-E40

The ECSS family of standards is organized around families. In particular, the Engineering family has a dedicated number for software (40). The ECSS-E40 document (Space Engineering - Software) recalls the various software engineering processes and list requirements for these processes in terms of activities that have to be performed and pieces of information that have to be produced. ECSS-E40 does not address directly the coding standards, but requires that the coding standards are defined and agreed, at various levels of the development, between the customer and the supplier.

In particular, the selection of this XML standard could be the answer to the following requirements of the standard ECSS-E-40A (Space Engineering – Software, 13 April 1999):

5.2.2.1 System Requirements, Expected Output e): Identification of lower level software engineering standards that will be applied [RB;SRR]

5.3.2.11 Each supplier shall define the Software Engineering standards that he intends to follow for his application area (Expected Output b) [RB, SRR]

5.4.2.1 Software requirements, Expected Output i): identification of lower level software engineering standards that will be used [TS;PDR].

1.3 Document Overview

This document is intended to build on the output of the *Software Top-Level Architectural Design*, *Design of Software Items* and *Coding and Testing* phases (ECSS-E-40 terminology) and follows a "top-down" approach so that guidelines can begin to be applied as soon as detailed design of software items starts. Chapter 4 provides a summary of the general guidelines.

Subsequent chapters describe the specific guidelines to be applied to the production of XML and XML Schema code.

All rules (mandatory) and recommendations (optional) are numbered for reference purposes.

All rules and recommendations have a short title and an explanation. Many rules and recommendations are also followed by a rationale section justifying the application of the rule. Rules and recommendations may also contain examples showing how to apply them, or illustrating the consequences of not applying them.

All rules and recommendations are enclosed in boxes. Recommendations are printed in *italic type*. Throughout the document, source code examples and references in text are printed in `fixed width` font.

1.4 Glossary

Attribute – A name-value pair attached to the element's start tag.

Binding – Binding APIs ensure the integration of the data model of XML Schema with the data model associated with the code implemented in a certain programming language.

Child – Each XML element that hierarchically has (one) parent element.

Element – The building block of XML data (its name is enclosed between markup tags), naming an entity of the represented data model.

Namespace – A means of distinguishing between elements and attributes from different XML-vocabularies that have the same name; for instance, the title of a book and the title of a web page in a a web page about books.

Parent – An element hierarchically higher in rank than its child element.

Parser – A tool that checks XML data for its well-formedness. (syntactic checker).

Root – Every XML document has one element without a parent. This is the first element in the document that contains all other elements.

Russian doll design – A design pattern for XML schemas that corresponds to having a single box (type or element), with other boxes nested within, as deep as necessary to cover the whole design (boxes within boxes, like a Russian doll)

Salami slice design – A design pattern for XML schemas that disassembles the XML data file into its individual components. In the schema, each component is defined separately (as an element declaration) and then assembled together to build up the whole definition.

Schema – An "implementation of a model" that can describe the allowed contents of XML data/documents conforming to a particular vocabulary.

Stylesheet – An implementation allowing to give XML data a particular screen- or print layout.

Tag – Delimiter for element name using markup syntax ("<></>").

Validator – A tool that validates XML data according to its defining/constraining Schema (semantic checker).

Venetian blind design – A design pattern for XML schemas that decomposes the schema into a number of type definitions, that are integrated later on.

1.5 Acronyms

API Application Programming Interface

ASN.1 Abstract Syntax Notation

CASE Computer Aided Software Engineering

DBMS Data Base Management System

DHTML Dynamic HyperText Markup Language

DOM Document Object Model

DP Design Pattern

DTD Document Type Definition

ECSS European Cooperation for Space Standardisation

HTML Hypertext Mark-up Language

HTTP HyperText Transfer Protocol

ISO International Standards Organisation

JAR Java Archive

JAXB Java API for XML Binding

JAXP Java API for XML Processing

JDOM Java Document Object Model

OO Object-oriented

OSS Open Source Software

RDF Resource Description Framework

SAX Simple API for XML

SGML Standard Generalized Mark up Language

SQL Structured Query Language

SW Software

WSDL Web Service Description Language

XDBMS XML Data Base Management System

XHTML eXtensible HyperText Markup Language

XKMS XML Key Management Specification

XMI XML Metadata Interchange

XML eXtensible Mark-up Language

XSL eXtensible Stylesheet Language

XSLT eXtensible Stylesheet Language Transformation

Chapter 2
General project Guidelines

2.1 Introduction

The objective of this chapter is to define *good coding rules and recommendations* applicable to software projects in general, independent of the data and coding standards. These common sense rules and recommendations *help* to make projects successful, within time and budget, and contribute to the maintainability and quality of the code.

2.2 Project Rules and Recommendations

Recommendation 1: Document any deviations. Never break a rule without documenting it.

Rationale

The complete set of rules is intended to provide consistency across all XML and XML schema files but this assumes that there are no constraints upon the programming environment. This is obviously not always the case and compromises may be necessary in order to handle specific requirements or restrictions of the operating system, the tool environment, of third party code and libraries, etc.

It is important to make sure first that you understand why the rule exists and what the consequences are if it it is not applied.

If a rule is broken on a project-wide basis, the reason for breaking it should be clearly laid out in the project documentation. On the other hand, if the rule is only violated in a few specific places, this should be clearly documented at those places. Anyone who examines the code at a later point will then be able to see why the rule was broken. It is to be hoped that there will also be some review process in which deviations from the guidelines will be examined.

Recommendation 2: Use independent tools to provide additional warnings and information about the code.

Rationale

Programmers are strongly advised to make use of other tools, which provide additional heuristic checks for problem and non-portable areas of the code. Other tools such as those used to extract documentation and cross-reference listings from the code are also useful because they exercise the code in a different way and therefore provide a additional point of view when looking for potential problems.

Recommendation 3: Use appropriate verification tools to ensure that the code conforms to the rules and to catch potential problems as early as possible.

Rationale

The use of software tools, if available, for the automatic analysis of code to check for conformance to the rules may aid the programmer to find problems. For the particular case of XSD, testing widely deployed schemas with more than one validator could help to ensure the portability and standards compliance of said schemas.

Recommendation 4: Make sure that any code that you use for debugging purposes does not have side effects.

Rationale

Any debugging code must not affect the overall state of the rest of the system. It may only modify parts of the system which relate to the debugging code itself, but not the rest of the code. The state of the system immediately before the section of debugging code must remain unchanged after it has been executed. This ensures that the behavior of the system does not change when debugging code is added or removed.

Chapter 3
Introduction to the XML Standards

Since the inception of the XML specification, a wide variety of technologies have been introduced, that support, among others, the creation, generation, processing, and validation of XML data. This chapter provides not only an overview of many of such technologies, but discusses the main motivations, advantages, and disadvantages of using XML for data modeling and communication.

3.1 Benefits of the Use of XML

XML is generally aimed at improving commonality, interoperability, interfacing and information retrieval for a wide variety of applications. In this regard, some of the valuable characteristics of XML are:

- It is coupled with simple programming interfaces, leading to easy interoperability with a variety of software systems.
- It is flexibility and adaptable (simplified change management.)
- It offers a variety of customization possibilities.
- It is extensible (eases change management and further development)
- Allows for unambiguously specifying data structure, relations, and interdependence, which can be properly connected to data semantics.
- Facilitates uncoupling any presentation aspects from the underlying data model and representation.
- Allows powerful and easy searching, is manageable.
- Most XML-based technology is modular.
- It is excellent for data sharing (good interoperability).
    - Enjoys widespread acceptance as standard, is widely endorsed by the software industry (supported in terms of tools, staff and training) and open (no vendor lock-in, license-free),
    - Has very little optional features in the core specification available, a fact that reduces ambiguities and eases implementation.
    - Is portable (independent from hardware, operating system, language, object model, etc.)

Another benefits of XML can be summarized as follows:

- Reduction of Commercial Of-The-Shelf (COTS) SW interface and purchasing costs.

- Decoupling of the producers of the data from the users: streamlining the development process and containing its costs.
- Decoupling of the design constraints of two intercommunicating applications: this would favor separate procurement, testing and qualification of applications using, for example, different programming languages.
- XML is independent from the programs that use or generate it.
- XML provides additional meaning and context to applications using data.
- Several space projects in ESA (Herschel, Planck, Cryosat, etc.) have chosen or are considering XML as a standard for their data and data interfaces.
- XML is safe and secure: XML helps to solve the problems caused by ambiguities. XML technology has maximum error checking and malicious code and characters will be detected as well. Encryption tools are readily available.
- Is compatible with the networked world (WAN, LAN, GAN, the Internet, the Grid, the Semantic Web.)
- XML is the basis of the 2nd Generation Internet (Semantic Web)
    - Intelligent searching.
    - Ubiquitous.
    - Better software architecture: higher performance.
    - Applications associated with URL-links, bi-directional links, intelligent software agents.

3.2 Drawbacks of the Use of XML

As well as benefits, XML has some drawbacks that also have to be considered before choosing it as a solution:

- *Unfamiliar structuring of data.* people tend to think of information in terms of what they can see (tables, paragraphs, fonts, forms, and so on). Most of them (except maybe for computer specialists and other related professionals) have a hard time thinking in terms of abstract, structured information. Users may require, for example, extensive training to use specialized XML editing tools.
- *Verbosity.* XML documents are verbose but, although often their size is smaller than that of equivalent word processor or spreadsheet files. XML documents compress very well. Moreover, storage space and bandwidth on modern computers and networks make tiny size optimizations a waste of time. Nonetheless, in some specialized areas XML extra size might matter. for example:
    - Real-time data sampling.
    - Extremely high-speed network applications.
    - Embedded devices with limited storage.

- *Excessive abstraction.* XML information is very abstract compared to a file for a desktop-publishing program or a database. It is always necessary to add something else to do anything useful with the information such as a stylesheet. When information has only a single, non-XML destination and it does not need to be searched or indexed, XML's extra abstraction brings no particular benefit.
- *Bottlenecks.* Libraries implementing some popular XML processing interfaces, such as DOM and XSLT, can consume significant amounts of memory and processing time, a situation that could seriously affect the performance of a busy server dealing with many simultaneous sessions. Validating XML data against an XML schema may depend critically on the availability of a network connection and the performance of the network determines the performance of the process to a great extend. To avoid these problems, many XML specialists recommend using schemas only for authoring and testing, and avoiding them in production-grade XML systems.
- *Limited vendor and tool support.* There are many small and medium size vendors in the XML market but a lack of standardization makes vendors sell proprietary solutions or individual components only. Big vendors bring to the market rather add-ons to their existing products, instead of producing generally useful XML applications.
- *Specification glut.* The core XML specification is compact, but related core technology specifications, or specifications related to a specialized domain (such as the medical or the space domain) tend to be very numerous. All of these cause software designers trouble in order to decide which standards to follow. New standards are also often made in advance of any proven need or real implementations. Competition among standard groups has also led to much duplication.

3.3 Use of XML within ESA space projects

The following are some of the main applications of XML in the context of ESA space projects:

- Archiving.
- Reports.
- Logging.
- Storage of configuration data.
- Static data description (data dictionary).
- Implementation of file interfaces between systems.
- External interfaces and products.
- Internal interfaces (caution is recommended when using XML for internal interfaces in a project.)

3.4 Overview of the XML Technologies

The following figure provides an overview of the different XML technologies used at ESA:

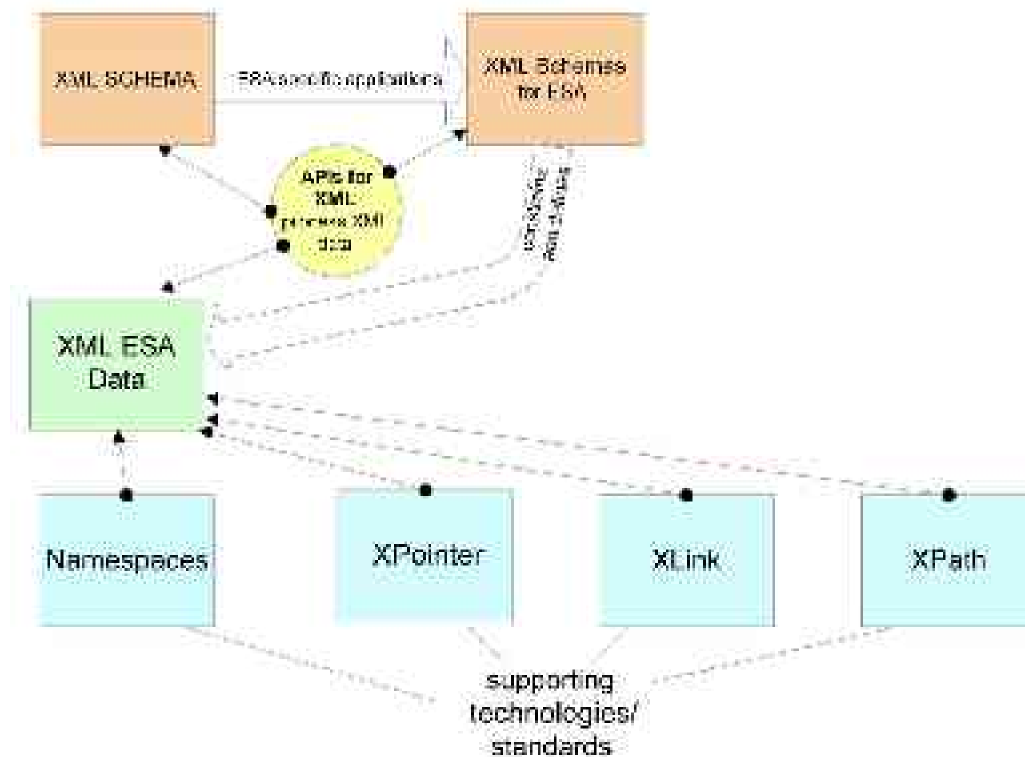

FIGURE 1: XML STANDARDS AND TECHNOLOGIES USEFUL IN ESA.

XML ESA Data

These are the actual data (e.g. sensor data, configuration data, telemetry data, logging information) that could be represented in XML format in the ESA context.

XML Schema

XML Schemas are written in XML and constrain the structure and format of XML data.

XSLT

XSLT makes it possible for data in a particular form of XML to be transformed to other forms of XML.

Namespaces

Namespaces have five purposes:

1. To distinguish between elements and attributes from different XML applications that share the same name (importing and including other schema documents).
2. To determine what elements and attributes schema declarations can validate.
3. To group all related elements and attributes from a single XML application together, so that software can recognize them easily.
4. In XPath expressions, used for *identity constraints*.

5. To reference global elements, attributes or types.

XPath, XLink and XPointer

XPath is a non-XML language used to identify particular parts of XML documents.

XPointer is a non-XML syntax used for locating points in, or ranges across XML documents.

XLink is an attribute-based syntax for attaching links to XML documents.

SAX

SAX, Simple API for XML, is a simple event-based API for parsing XML documents.

DOM

The Document Object Model is a W3C recommendation that describes a programming language neutral object model used to store hierarchical documents in memory.

JDOM

The Java Document Object Model is a recommendation that describes an object model used to store hierarchical documents in memory, optimized for the Java programming language.

JAXP

JAXP is the JAVA standard API for XML parsing and processing (applying XSLT to) XML

JAXB

JAXB is the Java standard API that binds/converts XML Schemas to Java classes and vice versa.

XML Databases

XML based DBMSs (XDBMS) are based on the hierarchical data model. The data themselves are stored as XML files on disk. The DBMS-functions (concurrency management, recovery, backup, restore, transaction management, data definition, data manipulation) are handled by tools and DBMS applications developed/provided by a commercial vendor, based on and developed with APIs that can process XML data and XML schemas.

3.5 Binary Encoding of XML

Several mechanisms (standards, protocols) and tools exist to ensure secure binary encoding of XML. This is, for instance, needed for high performance real-time transfers of data. Chapter 8 discusses this point in more detail.

3.6 XML Syntax Rules

This section contains a brief summary of the XML syntax rules.

3.6.1 Well-formed XML

As a minimum, all XML documents must be well formed (schemas can specify additional constraints). The rules for a well-formed document are as follows:

- Every start tag must have a matching end tag.
- Elements may not overlap.
- There must be exactly one root element.
- Attribute values must be quoted.
- An element may not have two attributes with the same name.
- Comments and processing instructions may not appear inside tags.
- No unescaped `<` or `&` signs may occur in the element's or attribute's character data.

3.6.2 Processing Instructions

Processing instructions are intended to pass information to XML processing tools. They are always enclosed in `<? ?>` symbols, for example:

```
<?xml version="1.0" encoding "ISO-8859_1" standalone="yes"?>
```

3.6.3 Comments

Comments are enclosed in `<!-- -->` tags. For example:

```
<!--This is the child element -->
```

Chapter 4
XML Tools and IDEs

Depending on its requirements, a project might require software tools to perform a number of possible XML related tasks. This chapter briefly discusses the main possible such tasks together with their corresponding tools.

4.1 Creation and Modification: XML Editors

XML editors allow to edit the element names, attribute names and comment fields of XML files.

4.2 Syntax Checkers: XML Parsers

XML parsers allow to verify XML files for syntax errors.

4.3 Semantic Checkers: XML Validators

XML validators allow to validate (for violation of definitions and constraints) XML data files (semantic validation) against schema files.

4.4 XML Browsers

XML browsers allow to navigate quickly through large XML files.

4.5 Publishing Software

Very often, it is necessary to convert raw XML to a format that is more comfortable for human readers to peruse. XML information intended entirely for machine to machine communication is, of course, excepted.

4.6 Archiving, Storage and Retrieval Software

Repositories, archives and shared directories are ways to store XML data for future use. Storage, location and retrieval software/tools handle the traffic of XML data to/from storage entities.

4.7 Security Tools

When XML documents contain secure or critical information, a project may need a system in place for signing, verifying, encrypting, and decrypting XML data.

4.8 Searching and Crawling Software

A *search engine* or a *crawler*, goes through shared directories and Web sites automatically, indexing any XML documents it may find. It enables the discoverability characteristics of the XML markup.

4.9 Distribution Software

If a project wants to deliver XML to many recipients, it may require some sort of syndication (publish/subscribe) system.

4.10 Routing and Auditing Software

If XML information goes through any kind of formal process where various people modify and approve it, a project may need some kind of workflow system to handle routing and maintain an audit trail.

4.11 Documentation Software

Humans working with XML need a way to discover the meaning of the markup, usually through technical documentation (Web-site, on-line help, etc).

4.12 Graphical IDEs

The ideal graphical IDE combines the capabilities of all types of XML tools and lets the user interact with the XML information by means of a friendly user interface. Many commercial tools exist, but their evaluation is outside the scope of this document.

Chapter 5
Naming

Using descriptive names for XML elements and attributes is as important as choosing proper identifiers in program code. An adequate name selection makes for data files that are easier to read, and for XML schemas that are easier to maintain.

This chapter is concerned with naming in XML files. The naming of XML files and directories (project-wide naming issues) are, however, outside of its scope.

5.1 General Naming Conventions

Rule 1: Use meaningful, familiar and consistent names.

Rationale

Descriptive names are preferable to *random,* arbitrary names. However, a descriptive name must reflect the actual use of the named entity over its lifetime. For example `count` is better than `xyz` for the name of an element which contains the total number of something, but not to name an element that holds an error code.

Recommendation 5: Avoid long (e.g. more than 20 characters) names.

While trying to keep them descriptive, avoid using very long names.

Rationale

Extremely long names may be hard to remember, are difficult to type, and may make XML code harder to format properly (not an issue in some circumstances).

Rule 2: Avoid making abbreviations for names, removing vowels.

Rationale

Abbreviations based on removing only bowels can be very hard to read.

Rule 3: In mixed-case names, capitalize the first letter of standard acronyms.

When using standard acronyms in identifiers, capitalize only the first letter, not the whole acronym, even if such acronym is usually written in full upper-case.

Example

Use `XmlFile`, `auxiliaryRmiServer` and `mainOdbcConnection` instead of `XMLFile`, `auxiliaryRMIServer` or `mainODBCConnection`.

Rationale

Doing this allows for clearer separation of words within the name, making identifiers easier to read. When only the first letter is capitalized, words are more easily distinguished without any one word being dominant.

Rule 4: Avoid names that differ only in case.

Never use identifiers in the same namespace that have the same letters, but that are capitalized in different ways.

Rationale

Similar names in a namespace are a source of potential confusion, that may lead to hard to detect errors.

Rule 5: Avoid using similar names, which may be easily confused or mistyped.

Avoid using names in the same namespace that differ by only in one character, especially if the difference is between `1` (the digit) and `l` (the letter) or `0` (the digit) and `O` (the letter). Avoid names that consist of similar meaning words because it is easy to confuse them. Consider, for example, the differences between `x1` and `xl` and between `countX` and `numberX`.

Rationale

Similar names in a namespace are a source of potential confusion, that may lead to hard to detect errors.

Recommendation 6: Names containing abbreviations should be considered carefully to avoid ambiguity.

Rationale

In the absence of a list of standard abbreviations, different programmers are likely to choose different abbreviations to represent a particular word or concept. This is especially true in the multi-lingual environment found in many ESA projects. Abbreviations should be used with care. For example, does `opts` refer to "options" or the "old points" or something else?

5.2 XML Data

5.2.1 Element Names

Rule 6: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in an element name.

Rationale

The capitalization provides a visual cue for separating the individual words within each name.

Example

```
bookPart
computerTime
value
```

Recommendation 7: Use nouns to name elements.

Rationale

Elements in data files usually represent parts of the structure of the corresponding data. As in a natural language it is adequate to name this parts using nouns.

5.2.2 Attribute Names

Rule 7: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in an attribute name.

Rationale

The capitalization provides a visual cue for separating the individual words within each name.

5.3 Namespaces

Recommendation 8: When using a standard namespace use the namespace prefix recommended by the corresponding schema author. Conversely, avoid using common prefixes when you are not referring to the namespace usually associated to them.

For common used namespaces, like those defined by W3C specifications, a namespace prefix is usually recommended, that should be used in conjunction with the namespace in question. Whenever possible, use such prefixes as recommended.

Rationale

Although an XSD schema could connect any prefix to a particular namespace, using the recommended prefix makes data files and schemas easier to read.

Chapter 6
XML and XML Schema Design and Style

6.1 XML Tree Design

Recommendation 9: XML trees should have the right level of depth and of width.

Rationale

Trees (hierarchies) that are too deep will make the data structure difficult to maintain but offer the advantage of good reuse. Trees that are too wide are more difficult to understand and modify but offer a better abstraction of the data.

Rule 8: Make two identical copies of all your schemas, where the copies differ only in the value of `elementFormDefault` (in one copy set `elementFormDefault="qualified"`, in the other copy set `elementFormDefault="unqualified"`).

Rationale

If you make two versions of all your schemas then people who use your schemas will be able to implement two design approaches: hide (localize) namespaces, or expose namespaces.

Hide the namespaces of the elements and attributes within the schema:

- When simplicity, readability, and understandability of instance documents is of utmost importance or when namespaces in the instance document provide no necessary additional information. In many scenarios the users of the instance documents are not XML experts.

  Namespaces would distract and confuse such users, where they are just concerned about structure and content.

- When you need the flexibility of being able to change the schema without impact to instance documents. To see this, imagine that when a schema is originally designed it imports elements/types from another namespace. Since the schema has been designed to hide (localize) the namespaces, instance documents do not see the namespaces of the imported elements. Then, imagine that, at a later date, the schema is changed such that instead of importing the elements/types, those elements and types are declared/defined right within the schema (inline).

  This change from using elements/types from another namespace to using elements/types in the local namespace has no impact to instance documents because the schema has been designed to shield instance documents from where the components come from.

Design the schema to *expose* namespaces in instance documents:

- When the lineage or ownership of the elements are important to the instance document users (such as for copyright purposes).
- If there are multiple elements with the same name but different semantics, then you may want to qualify them with namespaces so that they can be differentiated (e.g, `publisher:body` versus `human:body`). In some cases, there are multiple elements with the same name and different semantics, but the context of the element is sufficient to determine its semantics.

  For example, the title element in `<person><title>` is easily distinguished from the title element in `<chapter><title>`. In such a case there is less justification for designing your schema to expose the namespaces.
- When processing (by an application) of the instance document elements is dependent upon knowledge of the namespaces of the elements.

Recommendation 10: Minimize the use of global elements and attributes.

Rationale

This way `elementFormDefault` can behave as an *exposure switch*. There are two requirements on an element for its *namespace* to be hidden from instance documents:

1. The value of `elementFormDefault` must be `unqualified`.
2. The element must not be globally declared. For example:

```
<?xml version="1.0"?>

<xsd:schema ...>
  <xsd:element name="spacecraft">
  ...
</xsd:schema>
```

The element `spacecraft` can never have its namespace hidden from instance documents, regardless of the value of `elementFormDefault`. Spacecraft is a global element (i.e., an immediate child of `<schema>`) and therefore must always be qualified. To enable namespace hiding the element must be a local element.

6.2 Attributes and Child Elements

Rule 9: Uniquely identify all schema components with the `id` attribute.

This is *not* the same thing as creating an element with an attribute that has an `ID` data type. Rather, what is being referred to here is the capability to associate an `id` attribute with every schema component (types, elements, attributes, etc).

Rationale

This provides a finer level of granularity for identifying components than namespaces do, which only provide a coarse level of granularity.

Example

```
<xsd:element name="elevation" type="xsd:integer"
  id="flight:aircraft:elevation"/>

<xsd:complexType name="company" id="wrox:spacecraft:company"/>
```

Recommendation 11: Postpone decisions as long as possible: Postpone binding a type reference to an implementation, i.e., use dangling types.

Rationale

This helps to make schemas more reusable.

Example

In an `<import>` element the `schemaLocation` attribute is optional. Don't use it.

6.3 XML Types

Recommendation 12: When in doubt, make it a type. You can always create an element from the type, if needed.

Rationale

With a type, other elements can reuse that type.

Rule 10: If the item is not intended to be an element in instance documents, then define it as a type.

Rationale

Types are more general and thus, more reusable.

Example

If you will never see this in an instance document:

```
<instrument>
  ...
</instrument>
```

then define `instrument` as a `complexType`.

Rule 11: If the item´s content is to be reused by other items then define it as a type.

Rationale

Types can be reused as content, whereas elements cannot.

Example

If other items need to use `instrument`´s content, then define `instrument` as a type:

```
<xsd:complexType name="instrument">
   ...
```

```
</xsd:complexType>
...
<xsd:element name="radar" type="instrument"/>
<xsd:element name="gyro" type="instrument"/>
```

The example shows two elements, `radar` and `gyro`, reusing the `instrument` type.

> Rule 12: If the item is intended to be used as an element in instance documents, and it is required that sometimes it be nillable and other times not, then it must be defined as a type.

Rationale

Let us first see how not to do it. Suppose that we create an instrument element:

```
<xsd:element name="instrument">
  ...
</xsd:element>
```

The instrument element can be reused elsewhere by referencing it:

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

Suppose that we also need a version of instrument that supports a nil value. You might be tempted to do this:

```
<xsd:element ref="instrument" nillable="true"/>
```

This is not legal. This *dynamic morphing capability* (i.e., reusing a `instrument` element declaration while simultaneously adding nillability) cannot be achieved using elements. The reason for this is that the `ref` and `nillable` attributes are mutually exclusive: you can use `ref`, or you can use `nillable`, but not both.

The only way to accomplish the dynamic morphing capability is by defining `instrument` as a type:

```
<xsd:complexType name="instrument">
  ...
</xsd:complexType>
```

and then reusing the type:

```
<xsd:element name="instrument" nillable="true"
    type="instrument"/>
  ...
<xsd:element name="instrument" type="instrument"/>
```

In the first case `instrument` is nillable. In the second case it is not nillable.

> Rule 13: If the item is intended to be used as an element in instance documents and other elements are to be allowed to substitute for the element, then it must be declared as an element.

Rationale

Suppose that we would like to enable instance document authors to use interchangeably the vocabulary (i.e., tag name) `instrument`, `subSystem`, or `component`, i.e.,

```
<xsd:instrument>
  ...
</xsd:instrument>
...
<xsd:subSystem>
  ...
</xsd:subSystem>
...
<xsd:component>
  ...
</xsd:component>
```

To enable this *substitutable tag name capability,* `instrument`, `subSystem`, and `component` must be declared as elements, and made members of a `substitutionGroup`:

```
<xsd:element name="instrument">
  ...
</xsd:element>
<xsd:element name="subSystem" substitutionGroup="instrument"/>
<xsd:element name="component" substitutionGroup="instrument"/>
```

Recommendation 13: Do use attribute groups and model groups.

Rationale

Attribute groups, that is, a way to create a named collection of attribute declarations and attribute wildcards, increase the modularity of schemas. A commonly used set of attributes can be declared in a single location and then be referenced for other schemas.

A model group definition is a mechanism for creating named group of elements using the all, choice or sequence compositors. Model groups are useful for reusing groups of elements by avoiding type derivation.

Recommendation 14: Do use complex types and attribute declarations.

Rationale

A complex type definition is used to specify a content model consisting of elements and attributes. An element declaration can specify its content model by referring to a named or anonymous complex type. Named complex types can be referenced by name from the schema they are defined in or by external schema documents; anonymous complex types must be defined within the declaration for the element which uses the type. Anonymous complex types should only be used if there is no need for type derivation and if references to the type will not be needed outside the element declaration. Complex types are similar to model groups definitions with two differences. Firstly, complex types can include attributes in the content models they define. Secondly, it is possible to use type derivation with complex types.

Named complex types should be used rather than a combination of anonymous complex types, model group definitions and attribute groups for having the same capabilities. Using three mechanisms instead of one for specifying element content is more prone to confusion.

Recommendation 15: Use design by composition rather than subclassing.

Rationale

The advantages of the design by composition approach are:

- *Simplicity:* all of the components are stand-alone. They can be understood and developed independently, in isolation (separation of concerns, black box reuse).
- *Decoupled, changeable parts:* this approach focuses on creating a collection of independent, loosely coupled components. This enables robust, modifiable, plug-and-play designs.

Approaches to implement design-by-composition:

- *Direct containment:* the simplest form of design-by-composition is to simply embed an element declaration.
- *Containment by reference:* in this approach we embed an empty element with an IDREF attribute. The attribute references an element containing an ID attribute. This approach yields a very loosely coupled design.

Long, extended type hierarchies lead to brittle, non-modifiable designs that are virtually impossible to understand. Design by composition is the preferred approach. It yields simpler, robust, modifiable, plug-and-play designs.

Recommendation 16: If the decision is made to use subclassing, then avoid using derive-by-restriction.

Rationale

Using derive by restriction make the derived type must repeat the declarations in the parent type.

Example

```
<xsd:complexType name="t1">
  <xsd:sequence>
    <xsd:element name="e1" type="e1_type"/>
      <xsd:element name="e2" type="e2_type"
          maxOccurs="unbounded"/>
     <xsd:element name="e3" type="e3_type"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="t2">
  <xsd:complexContent>
    <xsd:restriction base="t1">
      <xsd:sequence>
        <xsd:element name="e1" type="e1_type"/>
        <xsd:element name="e2" type="e2_type"/>
        <xsd:element name="e3" type="e3_type"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Recommendation 17: If the decision is made to use subclassing, then limit the type hierarchy to maximum 3 levels.

Rationale

Understanding the type at the bottom requires understanding everything above it. Anything beyond 3 levels become unintelligible.

Rule 14: Do not use notations

Rationale

They exist only to provide backward compatibility with DTDs, except they are not backward compatible with DTD notations.

Recommendation 18: Do carefully use substitution groups.

Rationale

Substitution groups provide a mechanism similar to polymorphism in programming languages. One or more elements can be marked as being substitutable for a global element (also called the head element), which means that members of the substitution group are interchangeable with the head element in a content model.

The only requirement is that the members of the substitution group must be of the same type or be in the same type hierarchy as the head element.

Substitution makes content models more flexible but also processing of documents based on such schemas more complex. Members of a substitution group can be of a type derived from the substitution's group head. This is an extra complication since derived types can be of the restrictive type or of the extension type.

Rule 15: Do favor `key`/`keyref`/`unique` over `ID`/`IDREF` for identity constraints.

Rationale

Identity constraints are used for specifying unique values, keys, references to keys using XPath expressions defined within the scope of an element declaration and should have the preference over the use of the attribute's type `ID`, `IDREF`.

`ID`/`IDREF` have several limitations that identity constraints have not:

1. `IDs` can only have a specific range of values.
2. The XML Schema family of `ID` types are not entirely compatible with the the DTD family of `ID` types
3. An `ID` or `IDREF` has to be unique within the document. The symbol space for unique `IDs` is the entire XML document, but for unique keys is the target scope of the XPath. This is important if uniqueness is needed in two overlapping value spaces with different scopes in the same XML document.

Rule 16: Do not use default or fixed values especially for types of `xs:QName`.

Rationale

The primary complaint against default and fixed values is that they cause new data to be inserted in the XML document after validation, thus changing the data.

The use of `xs:Qname` may lead to incorrect behavior as it has no canonical form.

Recommendation 19: Do use `elementFormDefault` set to `qualified` and `attributeFormDefault` set to `unqualified`.

Rationale

Elements or attributes with a namespace name are said to be *namespace qualified*. The default value of both attributes of the `xs:schema` element: `elementFormDefault` and `attributeFormDefault` is `unqualified`.

It is possible to override whether local declarations validate namespace qualified elements and attributes or not.

Leaving the value of `attributeFormDefault` as `unqualified` makes sense because most schema authors do not want to have to namespace qualify all attributes by explicitly prefixing them.

6.4 Namespace and Namespace Referrals

Recommendation 20: Do use XML namespaces as much as possible. Learn the correct way to use them.

Rationale

An appropriate use of XML namespaces makes it much easier to reuse existing schemas by combining and/or extending them.

Recommendation 21: If you intend your schema type definitions to be reused in a variety of contexts, do not give them a `targetNamespace`. This is the so-called chameleon design pattern.

Rationale

A schema without a target namespace can typically only validate elements and attributes without a namespace name. However, if such a schema is included in a schema with a target namespace, the included schema assumes the target namespace of the including schema.

This feature is typically called the *chameleon schema* design pattern, and it is useful for creating a reusable module of type definitions and declarations.

There is a problem with combining chameleon schemas with identity constraints. Although `QName` references to types, definitions, and declarations in the chameleon schema are coerced into the namespace of the including schema, the same is not done for XPath expressions used by `xs:key`, `xs:keyref`, and `xs:unique` identity constraints.

Consider the following schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="Root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person" type="PersonType"
            maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="PersonKey">
      <xs:selector xpath="person"/>
      <xs:field xpath="@name"/>
    </xs:key>
    <xs:keyref name="BestFriendKey" refer="PersonKey">
      <xs:selector xpath="person"/>
      <xs:field xpath="@best-friend"/>
    </xs:keyref>
  </xs:element>
  <xs:complexType name="PersonType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="best-friend" type="xs:string" />
        <xs:attribute name="name" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

If this schema is included in another schema with a target namespace, the XPath expressions in both the `key` and `keyref` will fail. In this specific example, the person element is in no namespace in the chameleon schema, but once included in another schema it picks up that target namespace.

The XPath expressions which match on a person without a target namespace will not work without signifying that they no longer work since processors are not obliged to ensure that path expressions in identity constraint actually return results.

The point is that it is not advisable to use identity constraints in chameleon schemas.

Rule 17: If you have a `targetNamespace`, make it the default namespace.

Rationale

Advantages of applying the combination of the recommendation and rule above:

Schemas which have no `targetNamespace` must be designed so that the `XMLSchema` components (`element`, `complexType`, `sequence`, etc) are qualified. This approach will work whether your schema has a `targetNamespace` or not. Thus, with this approach you have a consistent approach to designing your schemas: always qualify the `XMLSchema` components with a namespace.

Disadvantages of applying the combination of the recommendation and rule above:

If your schema is referencing components from multiple namespaces then for some references you will have to qualify the reference, whereas other times you will

not (namely, when you are referencing components in the `targetNamespace`). This variable use of namespace qualifiers in referencing components can be confusing.

6.5 Integrating XML Schemas

Sometimes XML Schemas have to be integrated that come from different companies or from different project or development teams. The following issues have to be addressed when such an integration is needed:

- Elements and attributes that carry the same content should have the same name in the merged data structures and namespaces, or links should be used to make the connection.
- Versioning issues and configuration control issues.

Rule 18: Do not hard code the identity of an imported schema.

Rationale

Suppose that you declare an element to have a type from another namespace, e.g.,

```
<xsd:element name="sensor" type="s:sensor_type"/>
```

Observe that `sensor_type` is from another namespace. Thus, this schema will need to do an `<import>`. Normally we see `<import>` elements with two attributes: `namespace` and `schemaLocation`. However, `schemaLocation` is actually optional. When you do specify `schemaLocation` then you are rigidly fixing the identity of a schema which is to provide an implementation for `sensor_type`. We can make things a lot more dynamic by not specifying `schemaLocation`. Instead, let the instance document author identify a schema that implements `sensor_type`. This creates a very dynamic schema. The type of the sensor element is not fixed, static. Thus we postpone binding the type reference (`type="s:sensor_type"`) to an implementation of the type as long as possible, i.e., until schema validation time.

6.6 Extensible Content Models for XML Schema's

Recommendation 22: Create extensible schemas. Do use wildcards to provide well defined points of extensibility. Use the `<any>` element.

We can put the `<any>` element specifically in those locations where extensibility is desired. Of course, multiple `<any>` elements can be used if many points of extensibility are necessary. With `maxOccurs`, it is possible to specify *how much* extensibility will be allowed.

6.7 Expressing Additional Constraints

Although the XML Schema Definition language is quite powerful, certain constrains on the syntax of XML document cannot be expressed using it alone. A number of existing approaches make it possible to express such additional constraints in a way that allows for validating them automatically:

1. Supplement with another schema language.
2. Write code to express additional constraints.

3. Express additional constraints with an XSLT/XPath stylesheet.

The following subsections discuss these approaches in more detail.

Recommendation 23: Take into account that XML schemas will not be able to express all of your business rules.

Rationale

The expressive power of the XML schema language is limited, and generally restricted to syntax. Express any additional business rules that cannot be expressed directly in a schema by using an additional suitable constrain language.

6.7.1 Supplement with Another Schema Language

During the last years, a number of special schema languages have been proposed, that are intended to allow for expressing constrains on documents in a more general and convenient way. A good example of such a language is *Schematron* [SCH].

Advantages:

- *Collocated constraints.* There is something very appealing about having all the constraints expressed within one document rather than being dispersed over multiple documents. This is a common property of specialized constrain languages.
- *Simplicity.* Many of the schema languages were created in reaction to the complexity and limitations of XML Schemas. Consequently, most of them are relatively simple to learn and use.

Disadvantages:

- *Multiple schema languages may be required.* Each schema language has its own capabilities and limitations. Multiple schema languages may be required to express all the additional constraints. For example, while *Schematron* is very powerful, it is not able to express all constraints. Also, since it does not have loops and variables, *Schematron* may force the developer to go through many contortions to express some assertions.
- *Yet another vocabulary.* There are many schema languages, each with its own vocabulary and semantics. How do you find a schema language with the capability to express your problem 's additional constraints? You have to take the time to learn each of the schema languages. Hopefully, you will find one that supports expression of your constraints. Even if relatively easy to learn and use, it always takes time to learn a new vocabulary with is own particular semantics.
- *Questionable long term support.* In most cases, special schema languages are created by a single author. These authors are often busy, very bright people, whose interests may move to something else. If that happens you may be left with a product which is no longer supported.

6.7.2 Write Code to Express Additional Constraints

It is always possible to use a standard programming language to write code that parses an XML file and check constrains on it.

Advantages:

- *Full power of a programming language.* The advantage of this option is that with a single programming language you can express all the additional constraints.

Disadvantages:

- *Not leveraging other XML technologies.* There are other XML technologies that could be used to express the additional constraints in a declarative manner, without going through the compiling, linking, executing effort.

6.7.3 Express Additional Constraints with an XSLT/XPath Stylesheet

An XSLT/XPATH stylesheet can be used effectively to check constraints. The general approach is to use the XSLT language to identify patterns that are nor allowed and report them.

Advantages:

- *Application specific constraint checking.* Each application can create its own stylesheet to check constraints that are unique to the application. It is possible to enhance the schema without touching it.
- *Core technology.* XSLT/XPath is a *core technology* which is well supported, well understood, and has lots of material written on it.
- *Expressive power.* XSLT/XPath is a very powerful language. Most, if not every, constraint that you might ever need to express can be expressed using XSLT/ XPath. Thus you do not have to learn multiple schema languages to express your additional constraints.
- *Long term support.* XSLT/XPath is well supported, and will be around for a long time.

Disadvantages:

- *Separate documents.* With this approach you will write your XML Schema document, then you will write a separate XSLT/XPath document to express additional constraints. Keeping the two documents synchronized often needs careful management.

6.8 Schema Versioning

Similar to many other types of software artifacts, XML schemas need to evolve over time. Sometimes, a new version of a schema can be kept *backwards compatible*, that is, instances of older versions are still valid instances of the new version. It is often the case, however, that keeping a schema backwards compatible may be too complex or expensive, and that, for this reason, new incompatible versions have to be created.

Properly identifying the schema versions in both the schemas themselves and their instances, can thus be very helpful to adequately manage schema evolution. The following rules and recommendations, as well as the additional material in this chapter, are concerned with schema version identification and how to do it effectively.

Rule 19: Capture the schema version somewhere in the XML schema.

Always make sure that proper version information is included in your schema.

Rationale

Capturing the schema version inside the schema provides an easy way to check for a schema version. Certain approaches to registering the schema version in the schema itself (see the subsections in this chapter for a detailed discussion) may also allow for automatic matching of the version identifiers in schemas and instances.

Rule 20: Include information in the instance data files, that makes it possible to determine which version (or versions) of the schema they are compatible with.

Always make sure that any instance data files contain information identifying the schema version (or versions) the data is meant to be compatible with.

Rationale

Capturing the schema version inside the data file provides an easy way to identify matching schemas. Certain approaches to registering the schema version in the schema itself (see the subsections in this chapter for a detailed discussion) may also allow for automatic matching of the version identifiers in schemas and instances.

Rule 21: Make older versions of your XML schema available.

Rationale

This makes it possible for the users of the schema to keep relying in previous versions before they are able to migrate to the newer, potentially incompatible ones.

Rule 22: In situations where a new version of a schema makes backwards incompatible changes (e.g., a construct that was valid and meaningful for the previous schema does not validate against the new schema), make sure that older instances will not be accidentally validated against the new version.

Rationale

By properly identifying the schema version in the schema, it is possible to prevent older instances from being validated against a new version without noticing that the new version is backwards incompatible. If versions are properly identified, validation tools will immediately notice the incompatibility an report it, instead of producing potentially confusing error messages about some structures not being valid.

Example

The following steps (that correspond to the approach presented in Section 6.8.4) are a possible way to achieve the effect discussed above:

1. Change the target namespace. This will prevent older instances from validating at all.
2. Update the instances to reflect the new target namespace.
3. Confirm that there are no compatibility problems with the new schema.

4. Change the attribute that identifies the version/versions of the schema with which the instance is valid.
5. Update the schema file name/location if appropriate.

Recommendation 24: When an XML schema is only extended, (e.g., new elements, attributes, extensions to an enumerated list, etc.) one should strive to not invalidate existing instance documents.

For example, new elements or attributes could be made optional if this makes sense at all, since optional, additional elements or attributes will not break existing instances.

Rationale

Backwards compatible schemas reduce costs by avoiding unnecessary modification of instances.

Recommendation 25: Adopt a convention for schema version identification to indicate whether the schema changed significantly (changes were not backwards compatible) or was only extended (changes were backwards compatible).

One common schema is to use version identifiers consisting of a major and a minor version number, i.e., 1.0 or 2.3. The minor version number will be updated when changes to the schema are backwards compatible (i.e., version identifier goes from 1.0 to 1.1). The major version number, on the other hand, will be updated only after backwards incompatible changes (i.e., after backwards incompatible changes, version goes from 2.7 to 3.0).

Rationale

Having a convention that clearly distinguishes between backward compatible and incompatible changes, makes it easier to determine with which versions of a schema a given instance might be compatible.

6.8.1 Schema Versioning Techniques

As mentioned above, changes to a schema can be classified in two large groups:

1. *Backwards incompatible changes.* The new schema changes the interpretation of some element. For example, a construct that was valid and meaningful for the previous schema does not validate against the new schema.
2. *Backwards compatible changes.* The new schema extends the namespace (e.g., by adding new elements), but does not invalidate previously valid documents.

A number of options exist, that make it possible to identify the schema version in both schemas and their instances:

1. Change the (internal) schema version attribute.
2. Create a `schemaVersion` attribute on the root element.
3. Change the schema's `targetNamespace`.
4. Change the name/location of the schema.

The following subsections describe these alternatives in more detail, and compare their relative advantages and disadvantages.

6.8.2 Change the (Internal) Schema Version Attribute

In the first approach, one would simply change the number in the optional `version` attribute at the start of the XML schema. For example, in the code below one could change `version="1.0"` to `version="1.1"`

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="1.0">
```

Advantages:

- Easy. Part of the schema specification.
- Instance documents would not have to change if they remain valid with the new version of the schema (case 2 above).
- The schema contains information that informs applications that it has changed. An application could interrogate the version attribute, recognize that this is a new version of the schema, and take appropriate action.

6.8.3 Create a `schemaVersion` Attribute on the Root Element

With this approach, an attribute is included on the element that introduces the namespace. In the examples below, this attribute is named `schemaVersion`. This option could be used in two ways.

Usage A

First, this attribute could be used to capture the schema version. In this case, one could make the attribute required and the value fixed. Then each instance that used this schema would have to set the value of the attribute to the value used in the schema. This makes `schemaVersion` a constraint that is enforceable by the validator. With the example schema below, the instance would have to include a `schemaVersion` attribute with a value of 1.0 for the instance to validate.

```
<xs:schema xmlns="http://www.exampleSchema"
    targetNamespace="http://www.exampleSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
<xs:element name="Example">
<xs:complexType>
  ...
  <xs:attribute name="schemaVersion" type="xs:decimal"
      use="required" fixed="1.0"/>
  </xs:complexType>

  ...
</xs:element>
```

Advantages:

- The `schemaVersion` attribute is an enforceable constraint. Instances would not validate without the same version number.

Disadvantages:

- The `schemaVersion` number in the instance must match exactly. This does not allow an instance to indicate that it is valid using multiple versions of a schema.

Usage B

The second approach uses the `schemaVersion` attribute in an entirely different way. It no longer captures the version of the schema within the schema (i.e., it is not a fixed value). Rather, it is used in the instance to declare the version (or versions) of the schema with which the instance is compatible. This approach would have to be done in conjunction with option 1 (or an alternative indicator in the schema file to identify its version).

The `schemaVersion` attribute's value could be a list, or a convention could be used to define how this attribute is used. For example, if the convention was that the `schemaVersion` attribute declares the latest schema version with which the instance is compatible, then the example instance below states that the instance should be valid with schema version 1.2 or earlier.

With this approach, an application could compare the schema version (captured in the schema file) with the version to which the instance reports that it is compatible.

Sample schema, that declares its version as 1.3:

```
<xs:schema xmlns="http://www.exampleSchema"
    targetNamespace="http://www.exampleSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="1.3">

  ...
  <xs:element name="example">
    <xs:complexType>
      <xs:attribute name="schemaVersion" type="xs:decimal"
          use="required"/>
    </xs:complexType>
  </xs:element>
  ...
```

Sample Instance, declaring it is compatible with version 1.2 (or 1.2 and possibly other versions depending upon the convention used):

```
<example schemaVersion="1.2"
    xmlns="http://www.example"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.com/example.xsd">
  ...
</example>
```

Advantages:

- Instance documents may not have to change if they remain valid with the new schema version (case 2).
- Like option 1, an application would receive an indication that the schema has changed.

- Could provide an alternative to `schemaLocation` as a means to point to the correct schema version. This could be desirable where the business practice requires the use of a schema in a controlled repository, rather than an arbitrary location.

Disadvantages:

- Requires extra processing by an application. For example, an application would have to pre-parse the instance to determine what schema version it should be compatible with, and compare this value to the version number stored in the schema file.

6.8.4 Change the schema's `targetNamespace`.

In this approach, the schema's `targetNamespace` could be changed to designate that a new version of the schema exists. One way to do this is to include a schema version number in the designation of the target namespace as shown in the example below:

```
<xs:schema xmlns="http://www.exampleSchemaV1.0"
    targetNamespace="http://www.exampleSchemaV1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
```

Advantages:

- Applications are notified of a change to the schema (i.e., an application would not recognize the new namespace).
- Requires action to assure that there are no compatibility problems with the new schema. At a minimum, the instance documents that use the schema, and schemas that include the relevant schema, must change to reference the new `targetNamespace`. This is both an advantage and a disadvantage.

6.8.5 Change the Name/Location of the Schema.

This approach changes the file name or location of the schema. This mimics the convention that many people use for naming their files so that they know which version is the most current (e.g., append version number or date to end of file name).

Disadvantages:

- As with option 3, this approach forces all instance documents to change, even if the change to the schema would not impact that instance.
- Any schemas that import the modified schema would have to change since the `import` statement provides the name and location of the imported schema.
- Unlike the previous options, with this approach an application receives no hint that the meaning of various element/attribute names has changed.
- The `schemaLocation` attribute in the instance document is optional and is not authoritative even if it is present. It is a hint to help the processor to locate the schema. Therefore, relying on this attribute is not a good practice.

6.9 Global Versus Local Definitions

A component (`element`, `complexType`, or `simpleType`) is *global* if it is an immediate child of `<schema>`, whereas it is *local* if it is *not* an immediate child of `<schema>`, i.e., it is nested within another component. The main issue is, when should an element or type be declared global and when should it be declared local?

Below is a snippet of an XML instance document. We will explore the different design strategies using this example.

```
<spacecraft>
  <name>Huygens</name>
  <company>EADS</company>
</spacecraft>
```

Rule 23: Where minimizing size and coupling of components is of utmost concern then use the so-called *Russian Doll* design.

Rationale

The Russian Doll design corresponds to having a single *box* (`element` or `type`), having other boxes nested within, which in turn have boxes nested within them, and so on. (boxes within boxes, like a Russian doll)

This design approach has the schema structure mirror the instance document structure, e.g., declare a `spacecraft` element and within it declare a `name` element followed by a `company` element:

```
<xsd:element name="spacecraft">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="company" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</element>
```

Russian Doll Design Characteristics

1. *Opaque content.* The content of spacecraft is opaque to other schemas, and to other parts of the same schema. The impact of this is that none of the types or elements within spacecraft are reusable.
2. *Localized scope.* The region of the schema where the `name` and `company` element declarations are applicable is localized to within the spacecraft element. The impact of this is that if the schema has set `elementFormDefault="unqualified"` then the namespaces of `name` and `company` are hidden (localized) within the schema.
3. *Compact.* Everything is bundled together into a tidy, single unit.
4. *Decoupled.* With this design approach each component is self-contained (i.e., they don't interact with other components). Consequently, changes to the components will have limited impact. For example, if the components within `spacecraft` change the impact will be limited since they are not coupled to components outside of `spacecraft`.
5. *Cohesive.* With this design approach all the related data is grouped together into self-contained components, i.e., the components are cohesive.

Rule 24: Where your task requires that you make available to instance document authors the option to use element substitution, then use the so-called *Salami Slice design*.

Rationale

With this design we disassemble the instance document into its individual components. In the schema we define each component (as an element declaration), and then assemble them together:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="company" type="xsd:string"/>

<xsd:element name="spacecraft">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="name"/>
      <xsd:element ref="company"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Note how the schema declared each component individually (`name`, and `company`) and then assembled them together (by referencing them) in the creation of the `spacecraft` component.

The Salami Slice design corresponds to having many separate boxes (`element` or `type`) which are assembled together (separate boxes combined together, just like salami slices brought together in a sandwich).

The Salami Slice design also results in creating reusable (element) components, but it has absolutely no potential for namespace hiding.

Salami Slice Design Characteristics

1. *Transparent content.* The components which make up `spacecraft` are visible to other schemas, and to other parts of the same schema. The impact of this is that the types and elements within `spacecraft` are reusable.
2. *Global scope.* All components have global scope. The impact of this is that, irrespective of the value of `elementFormDefault`, the namespaces of `name` and `company` will be exposed in instance documents.
3. *Verbose.* Everything is laid out and clearly visible.
4. *Coupled.* In our example we saw that the spacecraft element depends on the `name` and `company` elements. If those elements were to change it would impact the `spacecraft` element. Thus, this design produces a set of interconnected (coupled) components.
5. *Cohesive.* With this design approach all the related data is also grouped together into self-contained components. Thus, the components are cohesive.

The two design approaches differ in a couple of important ways:

- The Russian Doll design facilitates hiding (localizing) namespace complexities. The Salami Slice design does not.

- The Salami Slice design facilitates component reuse. The Russian Doll design does not.

> Rule 25: The so-called *Venetian Blind design* is the one to choose when your schemas require the flexibility to turn namespace exposure on or off with a simple switch, and where component reuse is important.

Rationale

This design facilitates hiding (localizing) namespace complexities, and facilitates component reuse. Consider the spacecraft example again. An alternative design is to create a global type definition that nests the name and company element declarations within it:

```
<xsd:complexType name="mission">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="company" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="spacecraft" type="mission"/>
```

This design has both benefits:

- It is capable of hiding (localizing) the namespace complexity of name and company, and
- It has a reusable `mission` type component.

The instance document has all its components bundled together. Likewise, the schema is designed to bundle together all its element declarations.

With the Venetian Blind design we disassemble the problem into individual components, as the Salami Slice design does, but instead of creating element declarations, type definitions are created.

Here's what our example looks like with this design approach:

```
<xsd:simpleType name="name">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="integral"/>
    <xsd:enumeration value="cluster"/>
    <xsd:enumeration value="galileo"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="mission">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="company">
  <xsd:sequence>
    <xsd:element name="name" type="title"/>
    <xsd:element name="acronym" type="name"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="spacecraft" type="name"/>
```

This design has:

- *Maximized reuse.* There are four reusable components: the `name` type, the `mass` type, the `company` type, and the `spacecraft` element.
- *Maximized the potential to hide (localize) namespaces.* Note how this has been phrased: "maximized the potential ...". Whether, in fact, the namespaces of `title` and `author` are hidden or exposed, is determined by the `elementFormDefault` *switch*.

The Venetian Blind design exposes these guidelines:

- Design your schema to maximize the potential for hiding (localizing) namespace complexities.
- Use `elementFormDefault` to act as a switch for controlling namespace exposure. If you want element namespaces exposed in instance documents, simply turn the `elementFormDefault` switch to "on" (i.e, set `elementFormDefault="qualified"`). If you don't want element namespaces exposed in instance documents, simply turn the `elementFormDefault` switch to "off" (i.e., set `elementFormDefault="unqualified"`).
- Design your schema to maximize reuse.
- Use type definitions as the main form of component reuse.
- Nest element declarations within type definitions.

Here are the characteristics of the Venetian Blind Design.

Venetian Blind Design Characteristics

1. *Maximum reuse.* The primary component of reuse are type definitions.
2. *Maximum namespace hiding.* Element declarations are nested within types, thus maximizing the potential for namespace hiding.
3. *Easy exposure switching.* Whether namespaces are hidden (localized) in the schema or exposed in instance documents is controlled by the `elementFormDefault` switch.
4. *Coupled.* This design generates a set of components which are interconnected (i.e., dependent).
5. *Cohesive.* As with the other designs, the components group together related data. Thus, the components are cohesive.

If the developer wants namespaces exposed in instance documents:

- What if at a later date you change your mind and wish to hide namespaces?
- You will need to redesign your schema (possibly scraping it and starting over).

The solution is to adopt the Venetian Blind Design, which allows you to control whether namespaces are hidden or exposed by simply setting the value of `elementFormDefault`. No redesign of your schema is needed as you switch from exposing to hiding, or vice versa.

The particular project the reader is in may need to sacrifice the ability to turn on/off namespace exposure because you require instance documents to be able to use element substitution. In such circumstances the Salami Slice design approach is the only viable alternative.

Chapter 7
Security-Encryptions-Keys

Security APIs for Java, C and C++ are available in order to provide XML signature syntax and processing, XML encryption syntax and processing, and XML key management. Both commercial and open source, open standard implementations exist [XMLKEY].

Going into the details of the products is outside of the scope of this document as the product offer is currently quite volatile.

It is important to consider the W3C XML specifications on security see [XMLENC], [XMLSEC] and [XMLSIG].

Chapter 8
XML and Binary Data

8.1 Introduction

In its standard form, XML data will always be encoded as text. The application of XML to an always growing variety of purposes, has revealed a number of advantages, but also certain disadvantages, of XML's textual nature.

On the one hand, a text representation can be handled with relatively simple algorithms in a widely portable machine and architecture independent fashion. Additionally, XML text is also easily readable, thus making debugging and auditing applications much easier. On the other hand, the space overhead posed by the text representation could be pretty high for certain applications. It could even negatively affect the efficiency of processing algorithms due to the increased data throughput.

The idea of introducing and alternative *binary* encoding for XML that could overcome such limitations has been extensively, and quite often very fiercely discussed by the XML development community. Unfortunately, however, to the time of this writing no final solution has been found that could properly meet the complex and varied needs presented by various members of the community. Although many proposals have been discussed, none of them seems to really offer benefits that could compensate for the reduced simplicity and flexibility caused by the introduction of a binary representation.

The *XML Binary Characterization Use Cases* [XMLBINCASES] document, summarizes and effort by the *World Wide Web Consortium* (W3C) to collect a representative number of XML use cases, that could at some point lead to a better binary encoding for XML. This ongoing effort could eventually lead to one or more standardized solutions to the binary XML problem. In the mean time, practitioners must consider the advantages and disadvantages of the existing solutions and pick one based on their particular needs. This chapter presents a brief overview of the main existing options.

Recommendation 26: Consider your requirements carefully before deciding to look for a binary XML solution. In many cases, the standard text based XML encoding is adequate.

Rationale

A variety of trials an experiments performed by XML researchers and practitioners around the world [BOXPRJ, GIR, XMILL, XMLZIP] have failed to show actual benefits for a number of proposed binary encodings. The apparent inefficiency of XML's text encoding can be misleading, and an alternative binary encoding, that may result appealing in theory, could turn out to be equally or less efficient when tested in practice. If you still consider that a binary encoding may benefit your application,

conduct some performance tests to corroborate this fact before going on with the actual implementation.

8.2 Binary Attachments

Frequently, all what is needed is to introduce sets of binary data encoded in a well known binary format into regular XML files. An example of such a file could be a document containing bitmap pictures: The contents of the document could be represented as text, whereas the pictures would have to be encoded in a standard graphics format (JPEG, GIF, PNG) before embedding them in the document as binary data.

Depending on the size of the binary packages, different solutions are possible.

Recommendation 27: For embedding small amounts of binary data in an XML file, consider using a binary to text encoding land putting it directly in the XML stream.

Rationale

Small blocks of binary data can be easily converted to text and included in a regular XML file. A standard encoding like Base64 [RFC3584], which is supported by a variety of development environments, can be readily used for this purpose. [RFC2557] discusses this approach more in detail.

A disadvantage of Base64 and similar text encodings is that they increase the effective size of the data. An additional problem as that not all applications will automatically interpret the data as binary. For this reasons, refrain from encoding large amounts of data using this approach.

Recommendation 28: For packaging large amounts of binary data together with textual XML data in an efficient way, consider using the XML-binary Optimized Packaging (XOP).

Rationale

The *XML-binary Optimized Packaging* [XOP] combines the well known MIME standard for multi-part documents [RFC2557] with XML to allow for efficiently encoding large data sets containing a mixture of binary and text XML encoded data. Relaying on this standard may be the best way to guarantee interoperability with present and future applications.

8.3 XML Data Compression

In some cases, a more efficient encoding of data may be necessary, either to reduce space overhead (constrained storage media, slow communication links) or to increase data throughput. Data compression is a common approach to handle the space overhead.

Recommendation 29: Consider using a standard compression algorithm to compress XML data before storing or transmitting it.

Rationale

Modern compression algorithms are able to achieve very high compression ratios with a wide variety of XML data types. Additionally, most current development

platforms readily provide efficient, reliable, and standardized implementations of such algorithms. For these reasons, compressing XML data is often an efficient, practical and relatively well proven solution for most of the main drawbacks of the XML text encoding.

8.4 Other Binary Encoding Approaches

As mentioned, a number of binary encodings claiming better space usage or a higher data access efficiency have been developed during the last years. The effectiveness and maturity of such approaches is hard to assess in general, and, for this reason, the suitability for a particular project should be evaluated in the specific context of the project. A non-exhaustive list of binary XML approaches includes the BOX Project [BOXPRJ], the WAP binary encoding [WAP], the Millau encoding [GIR], and the XMill [XMILL] and XMLZip [XMLZIP] XML compressors.

Bibliography

ANS France Telecom. *ASN.1 Standard.*
`http://asn1.elibel.tm.fr/en/standards/index.htm`

ANSAPP
Application fields of ASN.1, web site, France Telecom,
`http://asn1.elibel.tm.fr/en/uses/index.htm`

ANSITU
International Telecommunications Union. *ASN.1 Project Web Site.*
`http://www.itu.int/ITU-T/asn1/`

ANSNOT
International Telecommunications Union. *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.* December 1997. `http://www.itu.int/ITU-T/studygroups/com17/languages/X.680_1297.pdf`

ANSSITE
France Telecom. *ANS.1 Information Site.* `http://asn1.elibel.tm.fr/`

ANSXML
France Telecom. *What ASN.1 can offer to XML.*
`http://asn1.elibel.tm.fr/xml`

ARM Armstrong, E. *Understanding XML..*
`http://java.sun.com/webservices/docs/1.0/tutorial/doc/IntroXML.html`

BOX Box, D. *Essential XML: Beyond Markup.* Addison-Wesley, 2000.

BOXPRJ
Binary Optimized XML (BOX) Project Web Site.
`http://box.sourceforge.net`

DODDIC
Dodds, L. *Dictionaries and Datagrams.* Published on XML.com, O'Reilly & Associates, January 2001.
`http://www.xml.com/pub/a/2001/01/24/deviant.html`

DODINT
Dodds, L. *Intuition and Binary XML.* Published on XML.com, O'Reilly & Associates, April 2001,
`http://www.xml.com/pub/a/2001/04/18/binaryXML.html`

DUB Dubuisson, O. *ASN.1 – Communication between Heterogeneous Systems.* June 2000. `http://asn1.elibel.tm.fr/en/book` and `http://www.oss.com/asn1/dubuisson.html`.

ECSS-E40-1B
European Space Agency (ESA). *Space engineering - Software (ECSS-E-40).* November 2003.

GIR Girardot, M., Sundaresan, N. *Millau: An Encoding Format for Efficient Representation and Exchange of XML over the Web.* In, *Proceedings of the 9th International World Wide Web Conference*, 2000
http://www9.org/w9cdrom/154/154.html

HAR Harold, E. R. *XML Bible Second Edition.* Hungry Minds, Inc. 2001.

JAVA
SUN Java Home Page. http://java.sun.com/

JAVAXML
Sun Microsystems' Java and XML Page. http://java.sun.com/xml/

LEV Levinson, E., Ed. *RFC 2387 The MIME Multipart/Related Content-type.* Internet Engineering Task Force (IETF), August 1998.
http://www.ietf.org/rfc/rfc2387.txt

MCLAU
McLaughlin B., *Java & XML,* O'Reilly & Associates.

NOK OSS Nokalva, Inc. *XML Support in OSS ASN.1 Tools, web site.*
http://www.oss.com/products/xml.html

OASIS
The Oasis Consortium Home Page.
http://www.oasis-open.org/home/index.php

OASISXML
OASIS XML Web Page. http://www.xml.org/

REIN Rein, L. *Handling Binary Data in XML Documents.* Published on XML.com, O'Reilly & Associates, July 1998.
http://www.xml.com/pub/a/98/07/binary/binary.html

RFC2557
Palme, J., Hopmann, A., Shelness, N. Eds. *RFC 2557 MIME Encapsulation of Aggregate Documents, such as HTML (MHTML).* Internet Engineering Task Force (IETF), March 1999.
http://www.ietf.org/rfc/rfc2557.txt

RFC3584
Josefsson, S., Ed. *RFC 3548 - The Base16, Base32, and Base64 Data Encodings.* The Internet Society, 2003.

ROS Rosen, D. *An Extensible Model for Real-Time XML Processing.* Presented at *XML Europe 2000*, June 2000.
http://www.gca.org/papers/xmleurope2000/pdf/s12-01.pdf

RUS Rusty H. et al. *XML in a Nutshell*, O'Reilly & Associates.

SCH Jelliffe, R. The Schematron: An XML Structure Validation Language using Patterns in Trees.
http://xml.ascc.net/resource/schematron/schematron.html

WAP Open Mobile Alliance (OMA). *WAP Forum Specifications.*
http://www.wapforum.org/what/technical.htm

XMILL *XMill: The XML Compressor.*
http://www.cs.washington.edu/homes/suciu/XMILL/

XMLBIN
World Wide Web Consortium (W3C). *Report From the W3C Workshop on Binary Interchange of XML Information Item Sets.* http://www.w3.org/2003/08/binary-interchange-workshop/Report.html

XMLBINCASES
Cokus, M. Pericas-Geertsen, S. *XML Binary Characterization Use Cases*. http://www.w3.org/TR/xbc-use-cases/

XMLBINGRP
XML Binary Characterization Working Group Public Page: http://www.w3.org/XML/Binary/

XMLENC
XML Encryption WG. http://www.w3.org/Encryption/2001/

XMLKEY
XML Key Management Specification. http://www.w3.org/TR/xkms/

XMLSEC
XML Security Apache Web Page. http://xml.apache.org/security/

XMLSIG
XML Signature WG. http://www.w3.org/Signature/

XMLSPEC
W3C XML specification Home Page. http://www.w3.org/XML/

XMLZIP
The XMLZip XML Compressor. http://www.sswug.org/see/XMLZip-9956

XOP Gudgin, M., Mendelsohn, N., Nottingham M., Ruellan, H. *XML-binary Optimized Packaging*. W3C Recommendation 25 January 2005. http://www.w3.org/TR/2005/REC-xop10-20050125/