OASIS Digital Signature Services (OASIS-DSS) An Architecture, Implementation and Interoperability

Master's Thesis at Graz University of Technology
2008

submitted by Konrad Lanz, bakk.techn., BSc Hons
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria
December 2008

Assessor: O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch
Supervisor: Ass.-Prof. Dipl.-Ing. Dr.techn. Peter Lipp

The candidate confirms that the work submitted is his own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)______________________

Abstract

OASIS Digital Signature Services (OASIS-DSS)
An Architecture, Implementation and Interoperability

This Master's Thesis investigates "OASIS Digital Signature Services (OASIS-DSS)", a client-server protocol that can contribute to hide the complexities of digital signature processing for "Extensible Markup Language (XML)" documents from the user. The challenges when creating "XML-Signature Syntax and Processing (XMLDSIG)" signatures range from making secure choices for transforms such as "XML Path Language Version 1.0 (XPath) filter transform", "Extensible Stylesheet Language for Transformation (XSLT)" to canonicalization (C14n) of XML. Further, various cryptographic signature schemes, key distribution and the secure choice of hash functions need to be taken off a client and are better manageable centrally.

When signed XML payload travels from client to server and back, broken signatures appear, the issue and proposed solutions are discussed in this thesis. When transforms and canonicalization in XMLDSIG depend on the context of the transport protocol signatures break. As such they are affected by inherited namespaces, inherited attributes or simply the presence of the transport protocol in the data model. XML documents can in general not be enveloped in one another.

This thesis proposes to amend XML itself to enable round tripping of signed XML data.

The architecture of OASIS-DSS itself was modified and extended to become implementable for all kinds of XML documents and transforms. A design and an implementation of an OASIS-DSS prototype library coping with the mentioned issues was performed.

During the work on this thesis standardization activities were undertaken in the "Organization for the Advancement of Structured Information Standards (OASIS)" and the "World Wide Web Consortium (W3C)". The author participated in the OASIS-DSS technical committee and W3C XML Security Working Group and its predecessor. Various amendments to OASIS-DSS addressing shortcomings identified during the work on this thesis have been committed.

Kurzfassung

OASIS Digital Signature Services (OASIS-DSS)
An Architecture, Implementation and Interoperability

Diese Masterarbeit untersucht „OASIS Digital Signature Services (OASIS-DSS)“ ein Client-Server Protokoll, das dazu beiträgt die Komplexität digitaler Signatur Verarbeitung von „Extensible Markup Language (XML)“ Dokumenten dem User abzunehmen. Die Herausforderungen beim Erstellen von „XML-Signature Syntax and Processing (XMLDSIG)“ Signaturen reichen von der Wahl sicherer Transformationen wie „XML Path Language Version 1.0 (XPath) Filtertransformationen“, „Extensible Stylesheet Language for Transformation (XSLT)“ bis hin zum Kanonisieren (C14n) von XML. Verschiedene kryptographische Signaturverfahren, Schlüsselmanagement und die sichere Wahl einer Hashfunktion können dem Client abgenommen werden. Diese Entscheidungen werden oft besser zentral auf einem Server getroffen.

Wenn signiertes XML vom Client zum Server und retour gesendet wird, brechen immer wieder Signaturen. Diese Problematik und Lösungsvorschläge werden in dieser Arbeit diskutiert. Auch wenn Transformationen und Kanonisierung in XMLDSIG vom Kontext des Transportprotokolls abhängen, können Signaturen brechen. Als solche sind diese von geerbten Namensräumen, geerbten Attributen oder schlicht von der Gegenwart des Transportprotokolls im Datenmodell betroffen. XML ist unter der Operation des Einfügens eines Dokumentes in ein anderes im Allgemeinen nicht abgeschlossen.

Diese Arbeit schlägt vor XML selbst anzupassen um XML uncodiert direkt in XML so transportieren zu können, dass signierte XML Inhalte originalgetreu erhalten werden.

Die Architektur von OASIS-DSS selbst wurde so modifiziert und erweitert, dass eine Implementierbarkeit für generelle XML Dokumente und allgemeine Transformationen möglich ist. Ein Design und eine Implementierung einer OASIS-DSS Prototypen-Bibliothek, die oben genannte Probleme behandelt, wurden durchgeführt.

Während dieser Masterarbeit wurden bei der „Organization for the Advancement of Structured Information Standards (OASIS)“ und dem „World Wide Web Consortium (W3C)“ Standardisierungstätigkeiten unternommen. Der Autor war Mitglied des OASIS-DSS Technischen Komitees (TC) und bei der W3C XML Security Working Group und deren Vorgängergruppe. An verschiedensten Änderungen von OASIS-DSS wurde fortlaufend mitgearbeitet.

Acknowledgements

I would like to thank . . .

Edith. 1998 was the luckiest year in my life, and not because XML was published. Edith thank you for your support, your patience and the last ten years.

Eva, my mum for supporting me in everything I ever did.

Peter Lipp my supervisor for trusting in my abilities right from the beginning.

The Institute for Applied Information Processing and Communications (*IAIK*) and its head Reinhard Posch my assessor for giving me the chance to do this work at this institute.

Many thanks to Harald Bratko for proof reading this document and Dieter Bratko for giving me advice on Public-Key Infrastructure (*PKI*).

Especially, Clemens Orthacker for his friendship and proof reading the document.

Konrad Lanz

Formatting conventions in this thesis.

- acronym (*ACR*) - acronyms
- *ACR* - abbreviated acronym
- `code` - in-line code elements
- <ds:Signature> - refers to Extensible Markup Language (*XML*) elements
- <dss:SignRequest> - refers to XML Schema (*Schema*) element declarations
- dss:SignRequestType - refers to *Schema* types declarations
- `RequestID` - *XML* attributes or *Schema* attribute declarations

Contents

Chapter 1

Introduction

Modern e-Government and e-Business processes rely on the strategic benefits of pure electronic transactions. Similar to paper based transactions, which can be signed and archived to provide evidence for the arbitration of potential disputes, electronic transactions need to be secured. The continuance of documents implies an End-to-End based approach as opposed to just securing a channel on a Point-to-Point basis. This means that transactions comprised of documents are secured and not just messages on their transmission channel. Signed documents, that can be stored and archived, allow to authenticate and protect the integrity of asynchronous remote communications by applying End-to-End [1] security principles.

The specification of OASIS Digital Signature Services (*OASIS-DSS*)[1] is driven by the need for web-based processing, which enables shared behavior of digital signature creation and verification across a group of users. To secure electronic documents technological decisions have to be made on an organizational level rather than by single users. Also other associated services like time-stamping are considered in *OASIS-DSS*, but are not the focus of this thesis. A web-based architecture avoids the need to implement the technical, as well as physical and procedural, complexities within user applications [2].

To support the processing of digital signatures the Organization for the Advancement of Structured Information Standards (*OASIS*) established a digital signature services technical committee (*TC*). It developed a protocol for digital signatures in a client server environment. The creation and verification of digital signatures via such a service facilitates the centralized control of the provision of signatures [3]. This thesis is focused on the *OASIS-DSS* protocol itself and its standardization, which also provides a set of basic elements to identify a service policy or a claimed identity accessing the service, nevertheless

[1]Terms written in *italics* and Acronyms (*ACR*) can be found in the Glossary starting at page 137 of this thesis.

access control, transport layer security, service policy rules and signature policies [4] are out of scope. Signature creation, signature verification, time-stamping and combinations of these comprise the list of services specified in *OASIS-DSS* [2]. Signatures facilitate to secure documents for asynchronous remote communications (Figure 1.0.1) whereas a *OASIS-DSS* server itself is to be accessed synchronously (Figure 1.0.2).

Computer supported cooperative work (*CSCW*) classification matrices

	synchronous	asynchronous
co-located		
remote		X

Figure 1.0.1: signed electronic documents

	synchronous	asynchronous
co-located		
remote	X	

Figure 1.0.2: *OASIS-DSS* itself

This document is structured into five chapters. A general introduction is given in chapter 1 followed by chapter 2 providing background knowledge about the Extensible Markup Language (*XML*) (section 2.1), its namespaces, types (section 2.2) and related technologies (section 2.3). Basics of digital signatures are discussed in section 2.4, which requires a working knowledge of applied cryptography. A good knowledge of *XML*, XML Schema (*Schema*) and XML-Signature Syntax and Processing (*XMLDSIG*) is expected in chapter 3. It introduces *OASIS-DSS* and contains comments on its structure as an outcome of the work on the *OASIS-DSS* standard. Then chapter 4 provides an analysis of *XML* in section 4.1, *XMLDSIG*, Canonical XML Version 1.0 (*C14n*) in section 4.2 and *OASIS-DSS* in section 4.3. In chapter 5 an architecture (section 5.1), a design (section 5.2) and an implementation of *OASIS-DSS* prototype library are given.

To put the contributions towards the *OASIS-DSS* standard into a chronological context it shall be noted that the work on this thesis started when *OASIS-DSS* was a working draft in revision 30 and the author of this thesis joined the *OASIS-DSS TC* in March 2005. Hence the analysis of *OASIS-DSS* (section 4.3) will address some of the shortcomings discovered in working draft 30 and the subsequent working drafts.

Note: The abbreviation DSS unfortunately matches the abbreviation used for another standard. To avoid confusion one has to appreciate that *OASIS-DSS* is not the same as Digital Signature Standard (*DSS*). When DSS or *OASIS-DSS* is mentioned in this document, then it refers to OASIS Digital Signature Services (*OASIS-DSS*). At this point another source for potential confusion shall be mentioned as well: the verbs verification and validation are often used interchangeably. Both terms are commonly used for describing the process of verifying a signature. *XMLDSIG* and Java Specification Request 105 XML Digital Signature APIs (*JSR105*) prefer the term validation whereas most pure cryptography related publications talk about verification of signatures [5]. This document shall also adhere to use the term verification for signatures, because the term validation is already used for validating an *XML* document against a Document Type Definition (*DTD*), *Schema* or similar grammar language.

Chapter 2

Background

This chapter provides the background knowledge necessary to analyze OASIS Digital Signature Services (*OASIS-DSS*), and begins in section 2.1 with an introduction to the Extensible Markup Language (*XML*) and how it can be used to create *XML* based languages. Then subsection 2.3.1 explains how *XML* is processed, subsection 2.3.2 shows how it is held in memory as a tree. How to address parts of *XML* is explained in subsection 2.3.3, subsection 2.3.4 and subsection 2.3.5 and finally how *XML* can be displayed can be found in subsection 2.3.6 and subsection 2.3.7.

XML and *XML* related technologies are discussed as they are an important basis to understand *XML* signatures and *OASIS-DSS*. In section 2.4 a short introduction and references to literature explaining the basics of digital signatures are provided. What XML signatures are and how they are processed is explained in subsection 2.4.1. Proficient in *XML* and related technologies, vested with a working knowledge about XML-Signature Syntax and Processing (*XMLDSIG*) and applied cryptography one can approach chapter 3 for an introduction to *OASIS-DSS*.

2.1 The Extensible Markup Language (*XML*)

Readers already familiar with Extensible Markup Language (*XML*) may just skim through this section or even skip it at all. Many people approaching *XML* for the first time however may note the hype still surrounding *XML* even ten years after its development. *XML* was standardized by a working group within the World Wide Web Consortium (*W3C*), started in 1996 and eventually issued the first Recommendation in February 1998. The overwhelming amount of related standards and huge amount of encompassing information offered today makes *XML* look to have lost its claimed simplicity by now. The purpose of this chapter is to demystify *XML* and to briefly explain its essentials.

XML in its simplest form can be viewed as a text document consisting of opening start tags (Figure 2.1.1 line 2) and closing end tags (line 8).

```
  <?xml version="1.0" encoding="UTF-8"?>
2 <documentElement msg="parent of firstChild and secondChild">
    <firstChild attr1="value">
4     This is in memory a text node and a child of element named firstChild.
      <mixedContent>is text and <elements/> mixed. This is</mixedContent>
6   </firstChild>
    <secondChild/>
8 </documentElement>
```

Figure 2.1.1: A simple well-formed XML document

These tags must be *properly nested* [6] and the opening tags can bear attributes (expression 2.1.2).

`attr1="value"` (2.1.2)

Between the tags one can find text[1], *processing instructions*, *comments* and yet again opening and closing tags. This form is referred to as the serialized *XML* representation, there elements are containers having a name and optionally attributes.
An element, which is an opening tag and a closing tag together, can also be seen to constitute a non terminal node, which is called an element node. One talks about nodes, because another important form of *XML* is its parsed tree representation as specified by the Document Object Model (*DOM*) (subsection 2.3.2) or the XML Path Language Version 1.0 (*XPath*) data model (subsection 2.3.3).
The attributes, text, *comments* and *processing instructions* are terminal leaf-nodes in *XML's* tree representation. Elements can also be leaf-nodes, when they are empty and do not have attributes (expression 2.1.3).

`<tagname/> or <tagname></tagname>` (2.1.3)

If an element has text and element children, we talk about an element having *mixed content* (Figure 2.1.1 line 5). A simple example of an *XML* document can be found in Figure 2.1.1.
XML is a subset of Standard Generalized Markup Language ISO 8879 (*SGML*) with some simplifications [7] and the intention to be rather human-readable ([6] section 1.1 Origin and Goals).
One can parse an *XML* document by using a recursive descent parser. To serialize *XML* depth first traversal over the tree representation emitting the serialized representations is performed. This is possible because *XML* must be *properly nested*.
XML's original intend is to mark up data which means to add meta data to text of a document. This meta data declares the types of the actual data and how its supposed to be processed or displayed, for instance as in Extensible HyperText Markup Language (*XHTML*). The actual presentation is not necessarily specified by the meta data itself, but may be specified by style-sheets written in languages

[1]In XML lingo this is called parsed character data or PCDATA.

like Cascading Style Sheets (*CSS*). They associate formatting information with the tags, so they can be displayed in a browser or printed.
In contrast to Hyper Text Markup Language (*HTML*), which has a predefined set of such meta data, *XML* allows for extensibility and to define the meta data itself. Thus *XML* allows to create new data formats or languages again based on *XML* . These are often written as normative text defining tags and attributes. Such specification usually defines the semantics, instructions to process[2] and sometimes also how to display newly defined tags. The latter as already mentioned is often performed by style-sheets or style-sheet transforms. Extensible Stylesheet Language (*XSL*) specifies how an instance of a class of *XML* documents is transformed into a displayable document like a Portable Document Format (*PDF*) or an *XML* document that uses a formatting vocabulary like *XHTML* (see Extensible Stylesheet Language for Transformation (*XSLT*) subsection 2.3.6). For a discussion about content and its presentation refer to [8] (section 2.2).
Although *XML* started as a pure markup language intended to publish documents like articles, books or electronic documents, it is used today for other data formats [9] and protocols like *OASIS-DSS* as well. It is already a popular file format for configuration files and data exchange between companies and applications.

2.1.1 The X and typing in *XML*

As *XML* claims to be human-readable and it may be typed using any text editor, the following section shall be concerned with typing in another sense and show how types and grammars work and what eXtensibility is in *XML.*
XML's extensibility does not only allow to define new tags, but also allows to extend existing data formats and languages. Extended document-instances containing newly defined tags should preferably remain processable by old applications, that ignore the new markup. It is assumed that the old markup can stand by itself. This is called the "Must Ignore" pattern of extensibility [10] enabling forwards compatibility. It potentially allows new documents to be processed in old applications, whereas backwards compatibility only requires new applications to be able to process old documents. Extensibility in the broader sense can however be understood as the ability to easily allow for updating a language. For instance the use of wild-cards as in XML Schema (*Schema*) plans for extensibility.
Normative text defining new or extended languages is mainly addressed at implementers and application developers and usually cannot be interpreted by machines and tools. To allow tools to validate document-instances against a grammar of an *XML* language specification, a formal grammar definition like a Document Type Definition (*DTD*) is needed and hence often also part of language specifications. *DTDs* specify which tags and attributes are valid in a document and how the tags form the structure of the document. Eastlake calls this to define "the allowable syntax of *XML*" [11]. The *DTD* is a grammar definition that lives directly in the Document Type Declaration (*DOCTYPE*) at the beginning of the *prolog* of an instance-document and is located behind the optional *XML* declaration (expression 2.1.4) and

[2]e.g. XML Property Lists

before the *document element* of an instance-document.

```
<?xml version="1.0" encoding="UTF-8"?>                    (2.1.4)
```

The *DTD* is comprised of an internal and an external subset. The external subset is referred to by a so called external identifier inside the *DOCTYPE*. The external identifier can either be a *system identifier*[3] directly referring to the resource or alternatively a *public identifier* specifying how the resource containing the *DTD's* external subset can be found. A *public identifier* is followed by system literal that works just like the system identifier and which is used as a default. A well known example for the latter is shown in Figure 2.1.5 used in *XHTML*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Figure 2.1.5: *DOCTYPE* declaration defines the document type for *XHTML* 1.0

A *system identifier* is converted to a *URI reference* that is referring to some file or data stream containing markup declarations. All markup declarations - external and local - taken together comprise a grammar called the *DTD*. Local markup declarations can extend the set of external markup declarations and attribute and entity declarations may even be overridden, although this functionality is very limited (cf. [6] "PEs in Internal Subset").
Figure 2.1.6 shows for example how to constrain *XHTML* to only allow level 1 and 2 headings.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
[
  <!ENTITY % heading "h1|h2">
]>
```

Figure 2.1.6: A local entity declaration overrides an external entity declaration

The *prolog* can further contain *comments* (expression 2.1.7), *processing instructions* (expression 2.1.8) and insignificant whitespace.

```
<!-- comment -->                    (2.1.7)
```

```
<?target content?>                    (2.1.8)
```

In Figure 2.1.9 one can see how those can potentially be arranged. We see an *xml declaration*, insignificant whitespace, a processing instruction and a *DTD*. The *prolog* ends with a comment and some more insignificant whitespace before the *document element*. There are more *processing instructions*, *comments* and insignificant whitespace, after the *document element* and then some elements have *mixed content*.

[3]Please recall that terms written in *italics* and acronyms (*ACR*) can be found in the Glossary starting at page 137.

```
<?xml version="1.0" encoding="UTF-8"?>
      <!-- comments followed by insignificant whitespace
         processing instructions -->
   <?ProcessingInstructionTarget content?>
<!DOCTYPE documentElement [
  <!ELEMENT documentElement (#PCDATA | firstChild | secondChild)*>
  <!ELEMENT firstChild (#PCDATA)>
  <!ELEMENT secondChild (#PCDATA)>
]>
    <!-- more comments-->
    <documentElement> A man has two children, one is called
       <firstChild> Alice
       </firstChild  > and the other one is called
<secondChild> Bob </secondChild>.
      </documentElement>
    <!-- more comments -->

   <?ProcessingInstructionTarget content?>

       <!-- more comments -->
```

Figure 2.1.9: A well-formed complex and untidy XML document

XML Schema (*Schema*) was introduced by the *W3C* as a replacement for *DTD*s [12] and is used more frequently today as part of a language specification. *Schema* is an XML vocabulary for describing the allowed contents [13] and structure of XML documents from other XML vocabularies.
Schema is not limited to create new languages from scratch. It allows to extend existing languages with new tags, attributes and constraints. This allows to augment, profile or restrict existing languages and should preferably be exercised in a forward and backward compatible fashion, thus preventing current implementations from breaking. Orchard discusses problems when extending existing languages [10] and introduces extension and versioning strategies.
Interestingly Extensible Markup Language 1.1 (*XML 1.1*) is not versioned in a forwards compatible fashion. Extensible Markup Language (*XML*) became an *W3C* recommendation on 10th of February 1998 and *XML 1.1* followed on the 4th of February 2004. *XML 1.1* differs mainly in addressing several requirements arising from internationalization like character sets and loosing the constraints on names for markup. Harold discourages the use of *XML 1.1* unless markup in languages not specified in Unicode 2.0 is required [14]. Orchard argues that *XML 1.1* extended *XML* where no such extension was allowed [10]. During our work for the XML Core Working Group (*XMLCORE*) the review of the fourth and fifth edition for *XML* and second edition of *XML 1.1* was on the agenda. It is likely that the compatibility defects will be fixed in a future edition by changing *XML's* name production. An *XML 1.1* document is then also an *XML* (1.0) document, despite its version number and differences with namespace undeclarations. Although we consider it fair and just to extend *XML's* character set for names (i.e. the tag and attribute names, do not discriminate certain user groups any more); it should be noted that *XML* may be changed in a way, that is not fully forwards compatible.
XML 1.1 is currently not of significance for this thesis as the *XPath* data model (subsection 2.3.3) is

currently not defined for *XML 1.1*[4]; changes to *XML* however are of interest as software produced for this thesis depends on libraries affected by such changes.

2.1.2 *XML* is a good choice for a signature and signed data format.

XML as such is not special and alternatives that make other trade-offs (section A.6) may pose a better choice than *XML* [16]; hence the following question should be asked: "What makes *XML* so popular and why is it so often preferred?"

It is the wide adoption in standards and tools that often makes *XML* the first choice, and it makes protocols, data formats, and configuration files less proprietary. One does not have to come up with new file formats over and over again, essentially reinventing already existing functionality. Today many document based applications like StarOffice, OpenOffice and MS-Office use *XML* and it is commonly used for platform independent data exchange. Protocols and languages based on *XML* are standardized and *XML* is today the driving technology for web services. Hence companies like Google, Amazon, Ebay and PayPal offer their services via Application Program Interfaces (*API*), many of which use *XML*. It currently is the state of the art for data exchange between loosely coupled components like in web services and service-oriented architectures, where data should be self-contained and to some extend self-explanatory.

Karlinger (section 1.1.1ff [17], in German) gives an overview of the history and advantages of *XML*. An important advantage mentioned, when signing *XML* or using it as a signature format, is that *XML* in contrast to Abstract Syntax Notation One (*ASN.1*) is comparatively human-legible and one does not require additional tools to read it. This however can be disputed for complex *XML* documents, where tools are required as well for comfortable reading. For relatively small, clean and simple *XML* documents however the assumption should hold true. The ability to check documents against a certain grammar allows to constrain *XML* documents and to keep them clean and as expected. Another advantage when signing *XML* was already mentioned by Scheibelhofer [8]. He points out that the signing of structured text like *XML* documents, separating structured data from its presentation, can allow visually impaired or hearing impaired user groups to choose an appropriate representation for the data to be signed or verified. It gives some degree of freedom to the presentation of the data.

It should be added here that *XML* technologies can even be an enabler to sign information as opposed to just signing data. Classes of *XML* documents to be signed can be accompanied by a *Schema* defining the allowed syntax. An additional description defining the semantics of the allowed elements, attributes and their interrelation complete a language definition. Such syntax and semantic definitions can be published and secured by signatures themselves. Hence a kind of closure to the process of signing information would be achieved using *XML* technologies consistently. An underlying assumption about common interpretation of how to define classes of *XML* documents and their semantics is required. This points into the area of standardization, standard compliance and signature policies [4], which are easier

[4]This may change if a new edition of *XPath* would be issued incorporating an erratum to section 5.4 [15].

enforced in a client server environment using an interface like *OASIS-DSS*. This thesis considers policies out of scope and they are mentioned here just for completeness.
XML technologies however are not the only way to achieve a similar effect. There is the socially highly accepted concept of paper documents that are mimicked for instance by *PDF* documents. It should be pointed out that such approaches accept the mixture of structure and presentation by relying on the analogy of representation on paper. Signing pure plain text documents like the average signed plain text email can also achieve the effect of signing information instead of just signing data by avoiding formatting all together. Nevertheless, a common interpretation and understanding about the character set[5] and its encoding has to be assumed (and potentially secured) for every text based data format (also for *XML*).
Pure plain text documents are in general not as visually attractive as a well designed markup document. Although depending on the application they may be sufficient and because of their simplicity even superior in certain use cases and for certain requirements.
To conclude, *XML* is not necessarily the best choice for each and every use case, but it seems to be a good choice for many use cases. It is an enabler to signing standardized information and aids interoperability. Its wide adoption and legibility aid self-containment and self-expressiveness of the data format.

2.2 Namespaces and Types

Having introduced *XML*, its extensibility and versioning in the previous section (subsection 2.1.1), this section will continue with Namespaces in XML 1.0 (*XMLNS*). They are used to avoid name conflicts and to allow for extensibility across different language specifications. *OASIS-DSS* makes use of *XMLNS* and so do other standards that are depending on *XMLDSIG* like for example XML Advanced Electronic Signatures (*XAdES*).

2.2.1 Namespaces help to extend XML grammars

The use of namespaces for various specifications and languages is supported by standardization bodies like *OASIS*, European Telecommunications Standards Institute (*ETSI*) and the *W3C* specifying grammars for standards in namespaces in their domain range. Within such a namespace a class of documents and their syntaxes and to various degrees also their semantics are defined. Such vocabularies are combined and can be joined to larger grammars.
As mentioned above *Schema* allows to define the syntax or grammar for a class of *XML* documents. *XMLNS* extends this functionality by allowing to join such grammars by prefixing the element and attribute names with a namespace prefix (eg.: `pb`) that has to be declared in a namespace declaration (expression 2.2.1).

```
xmlns:pb="http://example.org/b/"
```
(2.2.1)

[5]font substitution (2.2.1), `https://demo.egiz.gv.at/plain/projekte/dokumentenformate/pdf_a`

A namespace declaration looks like an attribute where the namespace prefix can be declared by a non colonized name (NCName) preceded by `xmlns` and separated by a colon. By definition `xmlns` is bound to `http://www.w3.org/XML/1998/namespace`. This namespace is reserved for this very purpose of declaring a namespace prefix.

```
  <?xml version="1.0" encoding="UTF-8"?>
2 <ee attr="has-no-namespace">
    <ea xmlns="http://example.org/a/" attr="has-no-namespace">
4     <pb:eb xmlns:pb="http://example.org/b/" pb:attr="in-namespace-b">
        <pc:ec xmlns:pc="http://example.org/c/" attr="has-no-namespace">
6         <ea1 xmlns:pa="http://example.org/a/" pa:attr1="in-namespace-a">
            <ed xmlns="" xmlns:pb="http://example.org/b2/">
8             <pb:ea2/>
            </ed>
10        </ea1>
        </pc:ec>
12    </pb:eb>
    </ea>
14 </ee>
```

Figure 2.2.2: An example showing the use of namespaces and namespace declarations.

The binding of a prefix to a namespace is in scope for the element bearing the namespace declaration and for all descendants. It is common to say, the children and its descendants inherit this binding. For the example shown in Figure 2.2.2, we say the namespace declaration for `http://example.org/b/` (line 4) in element `pb:eb` declares the prefix `pb` and is in scope for all descendants of `pb:eb` and for `pb:eb` itself. The comma separated notion of an element name is called a *prefixed name*. And the prefix is a shorthand for the namespace name it is declared for.

A default namespace can also be defined. It binds elements without a prefix to the default namespace by using a default namespace declaration . This is shown in Figure 2.2.2 and we say the namespace declaration for `http://example.org/a/` in element `ea` is in scope for all descendants of `ea` and for `ea` itself, despite `ed`.

`xmlns="http://example.org/a/"` (2.2.3)

XMLNS allows for a default namespace to be undeclared again by using a namespace undeclaration (expression 2.2.4).

`xmlns=""` (2.2.4)

By definition attributes without a prefix are not in a namespace (line 3, 5).

It is technically possible to redeclare the binding of a prefix to a namespace (expression 2.2.5, Figure 2.2.2 line 7). Documents using the same prefix for different namespace names can become confusing and hard to read. Also the interpretation of Qualified Names (*QName*) becomes harder.

`xmlns:pb="http://example.org/b2/"` (2.2.5)

There are two specifications defining namespaces, one for *XML* and the other one for *XML 1.1*. Their most significant difference is, that *XMLNS* allows to undeclare the default namespace only, whereas Namespaces in XML 1.1 (*XMLNS 1.1*) allow to undeclare namespace bindings, for any given prefix (expression 2.2.6). Wild-cards undeclaring a set or all of the of prefixes are not specified.

```
xmlns:prefix=""
```
(2.2.6)

The relation of the *XPath* data model to *XML 1.1* and also *XMLNS 1.1* is not well defined. This implies that *XMLDSIG*, because it uses the *XPath* data model, is also not well defined for *XML 1.1* documents. Nevertheless many implementations accept *XML 1.1* as input.

2.2.1.1 Qualified Name (*QName*)

XMLNS adds the term Qualified Names (*QName*) for all names that are subject to namespace interpretation. These are primarily element and attribute names and in Figure 2.2.2 all different kinds of *QNames* can be found. In line 2 `ee` is an element name that is in no namespace. In line 3 `ea` is an un-prefixed element name that lies in the default namespace, which however as mentioned earlier does not apply to attributes like `attr` in lines 3 and 5. As explained in the last section line 4 shows the *prefixed names* of an element (`pb:eb`) and an attribute (`pb:attr`). *QNames* however are not only used in element and attribute names, but also in content. Kay calls this namespace sensitive content (page 49 [18]).

2.2.2 XML Schema (*Schema*)

This section is meant to give a very brief and incomplete introduction to XML Schema (*Schema*) to be able to understand the essentials of *Schema* definitions for *OASIS-DSS* and *XMLDSIG*.
Schema is a language to define a grammar that *XML* documents can be - so to say - conforming to or valid against. In contrast to a *DTD Schema* does not have its own basic syntax, but it is *XML* itself. It also has a rich support for data types and is very verbose compared to grammar descriptions in Backus Naur Form (*BNF*) for instance.

> Grammar is a set of rules defining the structure of a family of XML documents. One type of grammar is the Document Type Definition (DTD) format defined by the XML specification. Another increasingly common type is the *W3C* XML Schema (Schema) format defined by the XML Schema specification. Grammars define which elements and attributes can be present in a document, and how elements can be nested within the document (often including the order and number of nested elements). Some types of grammars (such as Schema) also go much further, allowing specific data types and even regular expressions to be matched by character data content. [19]

Schema has support for data typing and is often a “type donor”[6] for other *XML* based languages, which themselves are often specified in terms of a *Schema* or *DTD*.

[6] cf. built in data-types Part 2 section 3 [20].

Schema supports simple and complex types, which are either defined in-line as *anonymous types* with local scope (lines 4-9, Figure 2.2.7) or they are named top level types (lines 12-16, Figure 2.2.7) with global scope.

```
   <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
     <xs:element name="MyInteger">
 4     <xs:simpleType>
         <xs:restriction base="xs:integer">
 6        <xs:minInclusive value="1"/>
          <xs:maxInclusive value="10"/>
 8       </xs:restriction>
       </xs:simpleType>
10   </xs:element>
     <xs:element name="MyShortString" type="MyShortStringType"/>
12   <xs:simpleType name="MyShortStringType">
       <xs:restriction base="xs:string">
14      <xs:maxLength value="3"/>
       </xs:restriction>
16   </xs:simpleType>
   </xs:schema>
```

An anonymous simple type and the simple top-level type named `MyShortStringType`.

Figure 2.2.7: Anonymous simple type and the simple top-level type

A *simple type* value is represented by character data content, if *XML* is in its serialized form. They may be restricted or checked against regular expressions, but a *simple type* can never contain attributes or other elements.

```
  <xs:element name="PointND" type="PointNDType"/>
  <xs:complexType name="PointNDType">
    <xs:sequence id="coordinates">
      <xs:element name="coordinate" type="xs:integer"
        minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="note" type="xs:string"/>
    <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>
```

Figure 2.2.8: The complex top level type named `PointNDType`

A *complex type* can have a structural description for attributes and elements and specify the order and quantity of the latter by means of sequences and choices. Attributes can only be of *simple type*. Unfortunately attributes are specified after the sequences or choices of elements in *Schema* which does not match their position in an *XML* document. There attributes appear in the start tag.

2.2.2.1 *Schema* Instance

An *XML* document can be associated with a *Schema* by means of attributes in the Schema instance namespace (expression 2.2.9), which is usually bound with the `xsi` prefix.

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```
(2.2.9)

The attribute `xsi:schemaLocation` allows for a space separated list, where namespaces alternate with the locations of the corresponding *Schema* resource. For elements and attributes that are in no namespace `xsi:noNamespaceSchemaLocation` is used.

Documents may also be validated against a *Schema* without such an association. Then the parser is instructed directly to use a certain *Schema* or dereference the namespace Uniform Resource Identifier (*URI*). A language specified in *Schema* is in the namespace provided by the `targetNamespace` attribute in the schema file. Which looks in the case of *XMLDSIG* as in line 31 of its *Schema* (Figure 2.2.10).

```
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
30         xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
           targetNamespace="http://www.w3.org/2000/09/xmldsig#"
32         version="0.1" elementFormDefault="qualified">
```

Figure 2.2.10: `targetNamespace` in xmldsig-core-schema.xsd

Schema offers a large set of built-in very expressive simple data types[7]. Minimal support of non-deterministic content models is provided by using `xs:all`. Otherwise it is very limited in its support for non-deterministic content models (chapter 7 [21]) and languages like RELAX NG, TREX and RELAX have better support in that area. It is the Unique Particle Attribution Rule (UPA-Rule) that forbids a *Schema* validator to look ahead to decide what branch of a grammar definition really matches. A very good example can be found in [10].
More advanced usage like a recursive *Schema* are not used in *XMLDSIG* or *OASIS-DSS*, however counter signatures in *XAdES* have a recursive character. Hence a short example for a recursive complex type shall be given in Figure 2.2.11, the complete schema with an example can be found in section A.3. Here in analogy to a file-system the complex type `FolderType` can contain `Folder` elements, which are themselves of `FolderType`.

2.2.2.2 Post Schema Validation Infoset (*PSVI*)

Schema and *DTD* can more than just defining and validating the allowed structure and value spaces of elements and attributes. They also allow to augment attributes with default values and type information, especially notable are here the attributes of type `ID` or `xs:ID`, whose type information is needed so

[7] `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#built-in-datatypes`

```
<xs:element name="Folder" type="fs:FolderType"/>
<xs:complexType name="FolderType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="fs:Folder"/>
    <xs:element ref="fs:TextFile"/>
    <xs:element ref="fs:BinaryFile"/>
  </xs:choice>
  <xs:attribute name="name" type="fs:folderAndFileNameType"/>
</xs:complexType>
```

Figure 2.2.11: recursive *Schema*

that they can be dereferenced. This has the drawback that a dependency from the *XML* document to the *DTD* or *Schema* is introduced and exceeds what the term validation covers. We hence conclude that the term validation is misleading as actual content is supplied via default values and type information is attached to the in memory representation.
The term Post Schema Validation Infoset (*PSVI*)[8] subsumes the set of information that is added to a parsed and validated in memory representation of *XML* and includes the latter.

2.2.2.3 *Schema* Patterns

There exists a set of state of the art *Schema* patterns that are often tacitly used and first have been documented by Costello [22]. They resulted from collaborative work by members of the "xml-dev" mailing list.

element declarations	type definitions	*Schema*-pattern
one global	all anonymous (local)	Russian Doll
	all named (global)	Venetian Blind
all global	all anonymous (local)	Salami Slice
	all named (global)	Garden of Eden

Figure 2.2.12: *Schema* Patterns

Many *Schemas* are however not using just one pattern, but rather mix some the patterns.

2.3 XML Technologies for Processing

In this section an overview is given, which technologies are used to access and process *XML*. The different layers for processing *XML* are explained from the bottom up. We start with the event based push parsing *API* called Simple API for XML (*SAX*) (subsection 2.3.1) and a short mention of the Streaming API for XML (*StAX*) is made; which is an *API* similar to *SAX*, but instead of being data

[8]Some parsers like *Xerces* allow to disable it using http://apache.org/xml/features/validation/schema/augment-psvi

driven it is a pull parsing *API*. Higher level *APIs* like the Document Object Model (*DOM*) or the *XPath* data model follow. They essentially represent the complete parsed *XML* document in memory as a tree and allow for navigational tree traversal and random access using attributes of type `ID`.
Most existing implementations of *XMLDSIG*, like the *IAIK* XML Security Toolkit (*XSECT*), are based on *DOM*, because the reference processing model of *XMLDSIG* requires random access as provided by *DOM*. *XPath* filtering requires navigational access and the *XPath* data model. *C14n* is defined on this data model as well.
DOM implementations often use *SAX* for parsing *XML* and then translate the events to nodes. They require a lot of memory to hold the documents. Future versions or profiles of *XMLDSIG* may constrain the reference processing model in a way, to make it suitable for streaming processing. Then pure *SAX* and *StAX* implementations of *XMLDSIG* could be created, which is why we consider the bottom up approach for explaining *XML* processing relevant. *CSS* used for presentation (subsection 2.3.7) and *URIs* addressing resources in the web and within documents are important basics for understanding *XMLDSIG*. To address certain parts of an *XML* document, *XPath* (subsection 2.3.3) expressions are an important tool. Other *XML* processing expressed in *XSLT* (subsection 2.3.6) is optional in *XMLDSIG*, but can be an enabler for rich presentation of actually signed *XML* content and a key to robustness.

2.3.1 Simple API for XML (*SAX*)

The Simple API for XML (*SAX*) is an event based push parsing *API* and it is part of the Java API for XML Processing (*JAXP*). When parsing an *XML* document, *SAX* fires events for every piece of *XML* it comes across. This means that a parser has to be set up with information about the input and handlers, which receive the events the *SAX* parser fires, are to be registered before the processing can begin. The events are categorized by the different handlers.

- *ContentHandler* - It receives events indicating the start and the end of a document and of elements. Attributes are included in the event for a start-tag. In between, events for character content, *processing instructions*, ignorable whitespace and the beginning and ending of namespace scopes are fired. It is notable that namespace scopes can start with the actual element declaring it, and are hence fired before.
- *LexicalHandler* - It receives events for *comments* and those indicating the start and end of *CDATA sections* with events for character content in between.
- *DTDHandler* - It receives basic events related to the *DTD* like notation declarations[9] and unparsed entity declarations.
- *Type Declarations Handler (DeclHandler)* - It receives events related to the *DTD* for element and attribute declarations as well as for external and internal entities.

[9] There is convention that notation declarations may be used to identify processing instruction targets, most code just compares target names as strings, rather than use DTDHandler.notationDecl() [23].

The ContentHandler is the most prominent interface as it handles the most important parts of an *XML* document: the start-tags including attributes, followed by character content, other start-tags and end-tags that close elements. A library that processes *XMLDSIG* however, will also have to be sensitive to *comment* and *CDATA sections*. Although the latter are not so critical, *CDATA sections* should preferably remain unchanged, because problems with the merging of adjacent text-nodes in the *XPath* data model can arise. This is why the LexicalHandler interface is almost as important for *OASIS-DSS* and *XMLDSIG* as the ContentHandler. The DTDHandler and DeclHandler are of less importance as the *DTD* is removed by *C14n*, entities are considered to be expanded beforehand and the *DTD* cannot be navigated in the *XPath* data model (cf. subsection 5.3.2). Nevertheless DTDHandler and DeclHandler affect the *PSVI* subsection 2.2.2.2, and are hence important if attributes of type `ID` are used in a document and by *XMLDSIG's* reference processing model. Attributes of the form *xml:id* mitigate this dependency as these are `ID` attributed in their own right [24], however only few parsers support it at this time.

To summarize *SAX* is intentionally centric around the ContentHandler to achieve simplicity. Hence registering other handlers with an *XML* parser like *Xerces* is rather cumbersome. Here setting the property for the LexicalHandler[10] and for the Type Declarations (DeclHandler)[11] is the only way and reflects that *APIs* more or less neglect other events.

SAX parsers are event producers and applications consume these events and either process them or transform them into data structures like a tree structure. *SAX* is also the underlying technology for higher level *API* implementations like *DOM*. Also stronger typed tree representations like data structures of Java Architecture for XML Binding (*JAXB*) that are based on some grammar like *Schema* build upon *SAX*. Applications using *SAX* directly however have to take care of maintaining context and data structures themselves. Purely *SAX* based applications can potentially be highly performance efficient in time and memory requirement, if applications optimize context and the data-structures held in memory. However as soon as random access within an *XML* document is introduced by attributes of type `ID`, referred to by others of type `IDREF`, memory efficiency suffers. The same is true when referencing in the same-document using fragment *URI* references as defined in XML Pointer Language Version 1.0 (*XPointer*) (subsection 2.3.5) like the shorthand (aka. bare-name) *XPointers*. Their random access nature leads to implementations that usually keep the complete parsed *XML* document as *DOM* in memory.

An *API* with similar properties is Streaming API for XML (*StAX*). It is a pull parsing *API*, which means the user or application iterates over the input, but it is as such not necessarily more efficient in time or memory usage than *SAX*. Brownell mentioned about the popular *SAX* implementations that neither *Crimson* nor *Xerces* include SAX-to-Text functionality [23], *StAX* offers this events to *XML* writing functionality. *StAX* gains importance as it is used more and more in web services, and claims to be easier to use than *SAX*.

[10]http://xml.org/sax/properties/lexical-handler

[11]http://xml.org/sax/properties/declaration-handler

2.3.2 Document Object Model (*DOM*)

The Document Object Model (*DOM*) represents *XML* data as an n-ary tree structure in memory. The nodes are connected by their parent-child relationships. The *root node* of this tree is the document node, which is not to be confused with the *document element* that is the first element node. All inner nodes are element nodes. The leaf nodes are empty element nodes, text nodes, nodes representing *CDATA sections*, comment nodes and nodes representing processing instructions.
All these nodes are ordered in their order of appearance in the document, and the first character of a node counts. This is the so called *document order* . The attribute nodes are conceptually not considered to be ordered within their owning elements, but could be also seen a set of leaf nodes attached to their owning element. An idiosyncrasy of *DOM* is that attribute nodes are not considered children of the element node owning them. This plays an important role when selecting parts of a document with *XPath* (subsection 2.3.3).
The document root node additionally contains data about the *XML* version, encoding and the *DOCTYPE*.

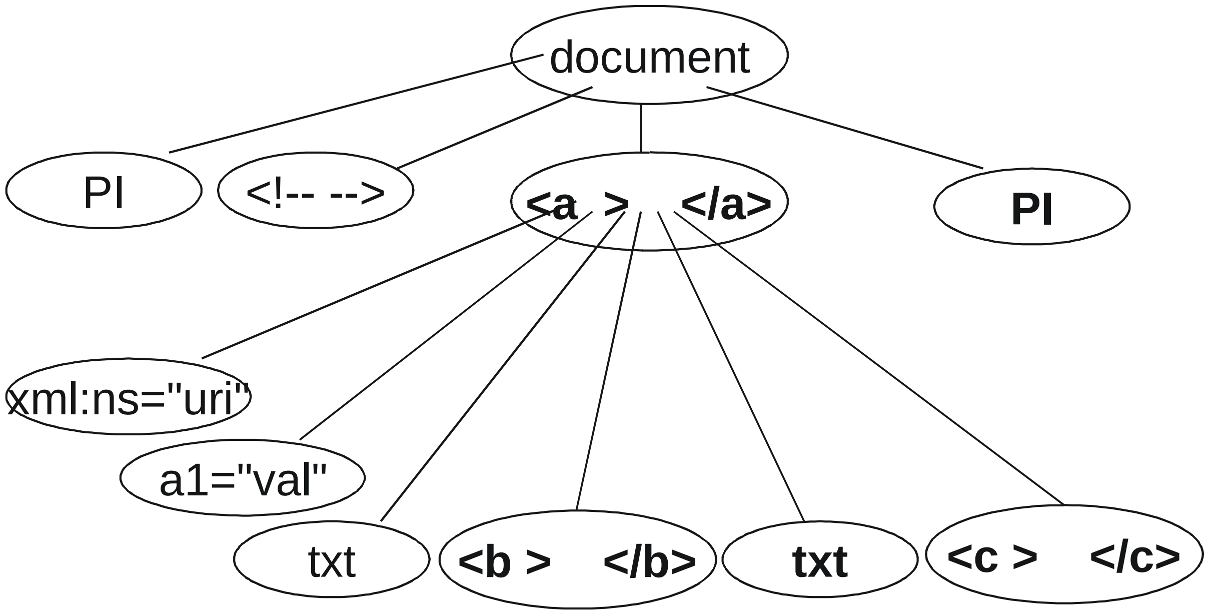


Figure 2.3.1: DOM Tree Representation

The biggest benefit of *DOM* is at the same time its biggest disadvantage: the complete document is held in memory. It allows for random access when resolving *same-document reference* and `ID` attributes. *XMLDSIG* requires random access within its document and hence currently all major implementations are based on *DOM*. An example is given in Figure 2.3.1 showing the complete spectrum of node types that may appear in a *DOM* representation of an *XML* document. In document order Starting with the conceptual document node the following nodes appear:

1. *document node* or *root node* (document)

2. *processing instruction* (PI)
3. *comment* node (`<!-- -->`)
4. *document element* (`<a> </a>`) owning attributes, that follow naturally (as a whole) the location of their owning element:
   - namespace declaration `xmlns:pre="uri"`
   - a normal attribute `a1="val"`
5. the content of the *document element* itself starts with a text node (txt)
6. followed by an empty element node (`<b></b>`)
7. yet another text node (txt)
8. and one more empty element (`<c></c>`).
9. eventually after the *document element* another *processing instruction* (PI) follows.

The order within the attributes of an element is of no significance by their definition in *XML*. Three things are worth noting in this example. First a document node may not contain text nodes and all whitespace located there is not reflected in the *DOM* representation as it is considered insignificant. Secondly there is only one document element and thirdly there may be *processing instructions* and *comments* before and after the document element.
The *DOM API* is versioned in levels, Level 3 is supported in *JAXP*1.3 or higher.
One state of the art library for parsing *XML* to *DOM* is *Xerces*, maintained as a Project by the Apache Software Foundation.

2.3.3 *XPath*

XPath is a non-*XML* syntax used not only by *XPointer* and *XSLT* for identifying particular pieces of XML documents, but also by *XMLDSIG* and XML encryption.
The *XPath* data model has a notion similar to that of *DOM* with a few exceptions and is of particular importance to *XMLDSIG* and *C14n*. A small difference is that the *document node* is called *root node* and a more important one is that namespace nodes are distributed to the child elements of the element bearing the namespace declarations and their children recursively (along the descendant axis). It should be noted that there are further more subtle distinctions between those data models. As signature breaks if only one single character or byte is represented differently, and one has to appreciate all intricacies of technologies underlying *XMLDSIG*. *DOM* for instance is aware of *CDATA sections*, while the *XPath* data model only knows text nodes and considers adjacent text nodes to be joined up to one text node[12].

[12]`http://www.w3.org/TR/DOM-Level-3-XPath/xpath.html#TextNodes`

Figure 2.3.2 shows the *XML* document corresponding to the *XPath* data-model in Figure 2.3.4.

```
<?xml version='1.0'?>
<?PI pival?>
<!-- comment -->
<a a1="1" a2="2" xmlns="a-ns" xmlns:pre="a-2-ns">
   <!-- comment -->
   <b/>
   <?PI pival?>
   <c c1="1" c2="2"/>
   <!-- comment -->
   <?PI?><c1/><d>
      <e/>
      <f/>
   </d>
</a>
<?PI pival?>
<!-- comment -->
```

Figure 2.3.2: XML document represented as *XPath* tree in Figure 2.3.4

In Figure 2.3.4 processing instruction nodes are indicated by `<?`, comment nodes by `<!--`, text nodes by `txt`, element nodes by a single character and the attribute nodes by their name in half the size of the element nodes. Namespace nodes are shown between the attribute nodes and the element nodes. Refer also to the legend in Figure 2.3.3. Although *XPath* does not make a distinction between nodes declaring a namespace and inherited namespace nodes all inherited namespace nodes are represented in light gray.

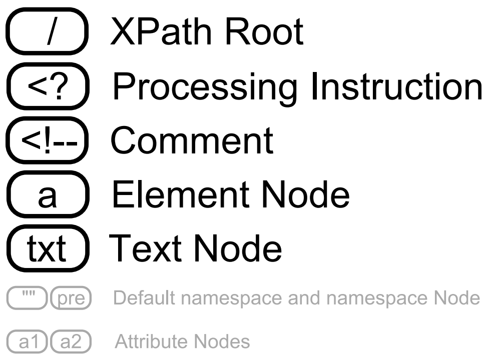


Figure 2.3.3: *XPath* Tree Representation Legend

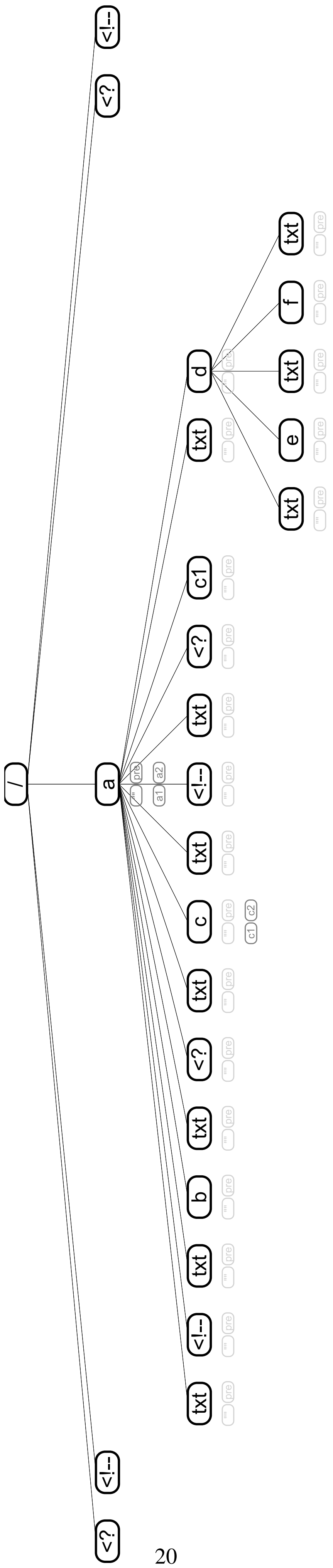


Figure 2.3.4: *XPath* Tree Representation for Figure 2.3.2

Earlier the descendant axis was mentioned, which contains all element nodes that are in a sub-tree excluding the element or document root that delimits the sub-tree. Axes are an important concept for *XPath* expressions, they define into which direction traversal through the tree shall go. Several axes are defined in *XPath* and they can be divided into four groups:

- self
  The self axis contains only the current context node.
- axes going laterally from the current context node
  - attribute
    The attribute axis contains only nodes if the context node is an element node having attributes.
  - namespace
    As potentially every node that is a descendant of the document element may have a namespace in scope the namespace axis contains nodes for all kinds of context nodes.
- axes running in document order
  - child
    The child axis contains the children of an element or document node and is empty for all other context nodes.
  - descendant
    The descendant axis contains all children and their children in depth first recursion (i.e. document order).
  - descendant-or-self
    This axis describes the same list of nodes as the descendant axis plus the actual context node.
  - following
    This axis describes in document order the list of all nodes that appear after the context node.
  - following-sibling
    This axis describes the set of all nodes having the same parent as the context node and appear in document order after the context node.
- axes running in reverse document order
  - parent
    The parent axis contains the parent of the context node thus is empty for the document node.
  - ancestor
    The ancestor axis contains context node's parent and its parent recursively in reverse document order and is empty for the document node.

– ancestor-or-self
  The same as the ancestor axis plus the context node.

– preceding
  This axis describes the set of all nodes that appear in reverse document order before the context node.

– preceding-sibling
  This axis describes the set of all nodes having the same parent as the context node and appear in reverse document order before the context node.

XPath is a query language used to select sets of nodes from an *XML* document. The example in expression 2.3.5 shows typical components of a location path, called location steps, that are separated by a slash '/'. For example expression 2.3.5 starts at the document root with a slash, then selects the *document element* if it is called `Signature` and is in the namespace associated with the prefix `ds` (usually the *XMLDSIG* namespace). From there the next location step spawns on all nodes of the node-set produced in the previous step. In this case going down the child axis the location path selects elements called Object in the same namespace having an attribute called `ID` with the value 'object1'.

$$\underbrace{\overbrace{/child::ds\colon Signature/}^{\text{location step}}\overbrace{child}^{\text{axis}}::\overbrace{ds\colon Object}^{\text{node test}}\overbrace{[@ID=\text{`}object1']}^{\text{filter step}}}_{\text{location path}} \tag{2.3.5}$$

A location step is composed of an axis, a node test and may be further narrowed by zero or more filter steps delimited by square brackets.
The result after a slash is a node-set and the next location step will be applied to all nodes of this node-set. This means the location step will be evaluated n times for a node-set of cardinality n taking one node after the other as context node. Note however that we will have a node-set after each location step and hence there will be no duplicate nodes. A good introduction to *XPath* can be found in one of Michael Kay's books (e.g. [18]).
XPath node-sets (aka. document sub-sets) are used as input for canonicalizing complete documents or parts of them. *C14n* defines its input for canonicalizing complete documents by node-sets according to expression 2.3.6 without *comments* and expression 2.3.7 with *comments*.

$$(//.|//@*|//namespace::*)[not(self::comment())] \tag{2.3.6}$$

$$(//.|//@*|//namespace::*) \tag{2.3.7}$$

Where the union (`|`) of all (`//.`) element, text, comment, and processing instruction nodes along the descendant-or-self axis is build with all attributes (`//@*`) along the descendant-or-self axis and with all the namespace nodes (`//namespace::*`) along the descendant-or-self axis.
In expression 2.3.6 a filter-step at the end removes all the comment nodes.

There are two versions of *XPath*, of which version 2.0 relatively recently became a *W3C* Recommendation (i.e. a Standard). *XPath* version 2.0 however does not play a role for the current version of *XMLDSIG*.

2.3.4 Uniform Resource Identifier (*URI*)

A Uniform Resource Identifier (*URI*) is defined in RFC3986 [25] and can be used as an identifier like a name or as a resource-locator like a reference or an address. The *URI* syntax is not discussed in detail in this section as it can be acquired from RFC3986 [25], RFC2396 [26] and other sources (section 7 [11]). *URIs* are normatively referenced in the relevant specifications and we discuss further how *URIs* are used in *OASIS-DSS* and *XMLDSIG*. *OASIS-DSS* directly refers to RFC2396 [26] and indirectly via *XMLDSIG*[27] as well as via *Schema's* `xs:anyURI` simple type.
XMLDSIG normatively refers to *URI* by referencing RFC2396 [26] and RFC2732 [28], both of which have been obsoleted by RFC3986 [25]. And RFC3986 [25] is referenced by XML Signature Syntax and Processing - Second Edition - (*XMLDSIG SE*)[29].

URIs as Identifiers, Names or Values

The use of *URIs* to denominate algorithms and cryptographic schemes is important and RFC3986 [25] defines the term identifier ...

> [...] to distinguish what is being identified from all other things within its scope of identification. Our use of the terms "identify" and "identifying" refer to this purpose of distinguishing one resource from all other resources.

Along with *OASIS-DSS Schema's* target namespace (expression 2.3.8) structured URNs (a subset of *URIs*) with a common prefix[13] (expression 2.3.9) are used for assigning well defined values to various inputs and messages (expression 2.3.10).

`targetNamespace="urn:oasis:names:tc:dss:1.0:core:schema"` (2.3.8)

`urn:oasis:names:tc:dss:1.0` (2.3.9)

`urn:oasis:names:tc:dss:1.0:resultmajor:Success` (2.3.10)

Similarly *XMLDSIG Schema's* namespace (expression 2.3.11) is used alongside with various *URIs* to identify algorithms (expression 2.3.12), qualify elements and specify encodings.

`targetnamespace="http://www.w3.org/2000/09/xmldsig#"` (2.3.11)

[13]prefix in the general sense, not namespace prefix.

`http://www.w3.org/2000/09/xmldsig#sha1` (2.3.12)

This usage of *URIs* as names is comparable to the use of string constants and they can be compared by their literal value, normalizations are usually not considered to be necessary.

URIs as References or address for Resources

The use of *URIs* as a reference or an address employs the term resource defined in [25] as

> [. . .] whatever might be identified by a URI. [. . .]

Which is to some extend circular and mainly lives form the examples provided:

> [. . .] an electronic document, [. . .] a service [. . .] human beings, corporations, and bound books [. . .] Likewise, abstract concepts [. . .], such as the operators, and operands of a mathematical equation, the types of a relationship [. . .], or numeric values [. . .].

This broad definition of a resource is useful to link from one resource to another and hence this document can link to an email addressee by using `mailto:Konrad.Lanz@iaik.tugraz.at`, to a telephone by using `tel:+433168730` (`callto:+433168730`), a Vo-IP connection like `skype:echo123`, for instance if read electronically.
However, when it comes to signing, *URIs* are per se not very useful and we have the additional requirement, that they need to be dereferenceable to an octet-stream, which can be digested and signed. This is commonly true for *URIs* using the `http:` scheme, when retrieving data from the web like in expression 2.3.13.

`http://www.example.org/index.html` (2.3.13)

Applications may even dereference *URIs* under the `tel:` scheme to binary data objects; for example by requiring the called party to answer by modem, using the dialpad or simply record the voice[14]. This is however not commonly used and such examples just serve the purpose of demonstrating that resource retrieval will have to be reproducible.
Such is necessary, so that verification can be performed at a later time especially as RFC 3986 *[. . .] does not require that a URI persists in identifying the same resource over time, though that is a common goal of all URI schemes [25].*
In the case of a phone call and recorded voice this becomes immediately apparent; two voice recordings of the same sentence will unlikely be binary equivalent data objects. This points to one of the immanent strenght of an *OASIS-DSS* request, where arbitrary data can be associated with arbitrary *URIs*. So the actual data retrieval is detached from the process of signing or verifying. Hence data stored during signing can be supplied for verification at a later time, which can be useful, if resource ceased out of existence or changed.

[14]*APIs* offer the possibility to extend resource retrieval for less frequently used *URI* schemes and dereference the resource.

Solutions for authentication and integrity assurance of the retrieval actions as such, as well as the potential need for confidentiality is out of the scope of this document and detached from *OASIS-DSS*. Evidence collection that such has been performed securely, may be modeled into application logic on top of *XMLDSIG* and *OASIS-DSS* to be potentially managed centrally for a group of users, but this would exceed the scope of this document.
XMLDSIG and *OASIS-DSS* live from the assumption that the data objects (i.e. the dereferenced resources) are reviewed by the signer before signing them (section 8.1.2 [27][29]).

The *relative URI reference*

As resources may also be identified relatively with respect to some base *URI* the term *URI references* should be used, because it encompasses both *URIs* and *relative URI references*. A *relative URI reference* is subject to interpretation against a base *URI*, called relative resolution (section 5.2 [25]). As relative resolution is idempotent, all *URI references* can be seen as being subject to relative resolution and we can be ignorant whether they are relative or not.
Examples of *relative URI references* include:

- `//www.example.org/`
- `//www.example.org/foo/bar/baz/index.html?boo=far#faz`
- `/foo/bar/baz/`
- `foo/bar/baz/index.html#faz`
- `../index.html?boo=far`
- `./../`
- `.././`
- `/`
- ./

Figure 2.3.14: Examples of *relative URI references*.

Same-document *URI* references

A *URI* reference can refer either within or to the same document bearing its lexical representation, which RFC 3986 [25] calls a *same-document reference*. All others are external *URI references*. These terms are important, because the processing in *XMLDSIG* and *OASIS-DSS* make use of them.

- same-document URI, which more accurately should be called a *same-document reference*, eg.:
  - `""` (the empty *URI reference*)
  - `#xpointer('/')`
  - `#element(/something/else)`
- external URI (reference) which is intended to be the opposite, eg.:
  - `http://www.example.org/path/file.ext#foo`
  - `../seg/`
  - `../seg/file.ext`
  - `file://c:/folder/file.ext#foo`

Figure 2.3.15: *same-document reference* vs external URI (reference)

XMLDSIG (section 4.3.3.2 - 4.3.3.3) uses the term *same-document reference* in a narrow sense limiting it to two idioms: the empty URI and fragments; but it has not brought up a definition. RFC2396 [26] (section 4.2) is unclear whether a *same-document reference* (Figure 2.3.16) is limited to those two idioms. RFC3986 [25] (section 4.4) mentions the two idioms, but conceptualizes even broader, requiring normalization and the availability of a base *URI* for comparison. So RFC3986 [25] may consider the *URI references* in Figure 2.3.16 as *same-document references*, as they will contribute either the empty string or only a fragment towards *URI* resolution. This however can depend on the path and potentially on the scheme.

- `foo/..`
- `foo/../`
- `bar`
- `foo/../bar#baz`
- `../foo/bar#baz`

Figure 2.3.16: Are those *same-document references*?

Nevertheless a base *URI* is required to be able to determine path and scheme. Only then can be said with certainty that we have a *same-document reference*. In the first two cases of Figure 2.3.16 the *URI reference* resolves to the empty string, which is a no-op and we can talk about a *same-document reference* assuming that the scheme has the usual syntax for the path components (eg. `http:`). Given a scheme the first two cases can be quite clear, but the last three cases are more subtle and every-

thing depends on the base *URI*. If the base *URI* ends in .../`bar` in the third and the fourth case or ...`foo/bar` in the last case, we potentially have *same-document references*. For example the base *URI* `http://example.org/foo/bar` with the following *URI reference* `../foo/bar#baz` results in `http://example.org/foo/bar#baz` which in this case identifies the same document, but may not in the case of another base.
To summarize, things become here very quickly very subtle when the resolution has to be considered detached from an actual retrieval action (e.g. on a local file system). RFC3986 [25] is not useful for a simple syntactical definition, and we even have not asked, whether `foo/` would have to exist in the first, the second and the third case, in the fifth it would.
Eventually in *XMLDSIG SE* [29] we have been explicit enough:

> In this specification, a 'same-document' reference is defined as a URI-Reference that consists of a hash sign ('#') followed by a fragment or alternatively consists of an empty URI [URI].

Which is the union of what RFC 3986 calls a fragment-only *URI* [25] and the empty *URI* reference (""). So we note that a fragment-only *relative URI reference* and the empty *URI* are technically also subject to interpretation against a base *URI*. They however do not depend on the scheme but may depend on the media type [25]. They can not result in leaving the current resource in *XMLDSIG*, because they do not have a path component and the knowledge of a base *URI* or its scheme is not required.

- "`#xpointer(/)`"
- "`#faz`"
- "`#xpointer(id('faz'))`"

Figure 2.3.17: Examples of fragment-only *URI* references.

It should be noted that a necessary assumption is, that an *XML* signature bearing the *same-document reference* is in a resource of an *XML* Internet media type (aka. MIME type or Content-type), which is necessary to have the semantics of the fragment defined. This seems to be a reasonable assumption for a *same-document references* inside an *XML* signature.
Let the following be what we will call the *XML* Internet media types:

- text/xml
- application/xml
- text/xml-external-parsed-entity
- application/xml-external-parsed-entity

For these the *XPointer* framework [30] defines the fragment semantics.
We can hence conclude that *XMLDSIG* and *OASIS-DSS* make a distinction between *same-document references* and all other *URI references* called external *URIs*.

The "Java Specification Request 105 XML Digital Signature APIs (*JSR105*)" implements the dereferencing of *URIs* in *XMLDSIG* by a special class to allow applications to use *URIs* in a flexible way. *OASIS-DSS* implementations will have to make use of custom *URI* dereferencing as it is done within the protocol for arbitrary *URIs*. The distinction between *same-document references* and all external *URIs* will have to be respected, when processing *OASIS-DSS*.

2.3.5 *XPointer*

XPointer [31] brings *XPath* and *URI* together. It is used to identify fragments or sub-resources by means of a *URI reference* locating a resource that is an *XML* document.

Shorthand *XPointers*

The most frequently used *XPointer* is at the same time one of the most common ways to reference sub-resources within the same document. It is called the shorthand (aka. bare-name) *XPointer* and is shown in expression 2.3.18.

`#someId` (2.3.18)

The original definition of bare-name *XPointers* "normative" for *XMLDSIG* was based on the *XPointer* candidate recommendation from 2001 [31] (section 4.2.2 [31]):

> [...] A bare name stands for the same name provided as the argument of `id()`. [...]

That in turn refers to *XPath's* `id()` function (section 4.1 [32]):

> [...] the result is a node-set containing the elements in the same document as the context node that have a unique ID equal to any of the tokens in the list. [...]

These references provide no direct indication about what should happen with unknown identifiers. Hence implementations may falsely assume that returning the empty node-set may be appropriate in such a case. Nevertheless when the *XPointer* specification is more carefully examined it should be noted that *XPointer* has a different goal from *XPath* and will not allow for empty results (section 3.4 [31]):

> [...] the *XPointer* language is intended as a specification of locations rather than a broader query language, an empty result is an error. [...]

XPointer until today never became a *W3C* recommendation and has not exceeded the status of a candidate recommendation. Hence it shouldn't be normatively referenced. It has been superseded by the

XPointer Framework [30], the `xmlns()` [33] and `element()` scheme [34]. The pure `xpointer()` scheme was relegated to being a working draft [35].
XMLDSIG SE recognizes that the *XPointer* candidate recommendation from 2001 [31] has never become a recommendation and refers now to the newer specifications [30], [33] and [34]. They are consistent with respect to empty results (section 1.2 and section 3.2 [30]) :

> [Definition: error] A violation of the syntactic rules of this specification, or the failure of a pointer to identify subresources. [. . .] If no element information item is identified by a shorthand pointer's NCName, the pointer is in error.

The bare-name *XPointers* are now called shorthand *XPointers* and *XMLDSIG SE* normatively references them in [30].

Full *XPointers*

The second *XPointer* idiom *XMLDSIG* mandates to be supported is a full *XPointer* either selecting the document root or an element by `ID`.

$$\texttt{\#xpointer(/)} \tag{2.3.19}$$

$$\texttt{\#xpointer(id('someId'))} \tag{2.3.20}$$

The paradox situation with the latter is that the `id()` function, although depicted in various examples of the XPointer Framework, will remain not to be normatively defined until *XPointer* would become a recommendation. *XPath* could not be referenced directly by *XMLDSIG SE* because its `id()` function returns the empty node-set as mentioned above. Hence the `element()` scheme was used to functionally specify it, the syntax however stayed the same and is now defined in *XMLDSIG SE*.
When we regard *XPointers* isolated, then expression 2.3.20 is equal to the shorthand *XPointer* in expression 2.3.18.*XMLDSIG* however introduces in section 4.3.3.2 an artificial distinction that requires *same-document reference* shorthand *XPointers* to dereference node-sets without comment nodes.
All other *XPointers* are discouraged in *XMLDSIG SE*, but remain optional and the *XPointer* candidate recommendation [31] remains to be their specification. They are however of significance as existing standards have adopted them already as we will see later.

External *XPointers*

External *URI* references even such as in expression 2.3.21 containing a fragment must be dereferenced to a stream of octets as required in *XMLDSIG* section 4.3.3.2 [29][27].

$$\texttt{http://www.w3.org/TR/xmldsig-core/\#xpointer(id('sec-Acknowledgements'))} \tag{2.3.21}$$

This plus the fact that neither a serialization of secondary resources nor an encoding for transmitting node-sets, nor how sub resources or node-sets are supposed to be parsed have been defined, implies for resources of some *XML* Internet media type that complete primary resources (the whole document) are to be dereferenced by any *XMLDSIG* application, which would be consistent with the Internet architecture (section 2.6 and 3.2.1 [36]). There it says:

> Interpretation of the fragment identifier is performed solely by the agent that dereferences an URI the fragment identifier is not passed to other systems during the process of retrieval. This means that some intermediaries in Web architecture (such as proxies) have no interaction with fragment identifiers and that redirection (in HTTP [RFC2616], for example) does not account for fragments.

At the time *XMLDSIG* was written, there was no consensus about that fragments do not cause the dereferencing of subresources, which has become good practice in 2003 and eventually normative in 2004 [36].
This explains, why the use of *XPointers* is not recommended in external *URIs*, as the interpretation of the fragment part is defined by the external resource's MIME type [36] (aka. Internet media type or Content-type) and the selection of the fragment may not be performed by the signature application.

XPointers degenerated by *XMLDSIG*

Looking at *XMLDSIG* (section 4.3.3.3 [27]) and *XMLDSIG SE* (section 4.3.3.3 [29]) one can see that *XPointer* in *XMLDSIG* is effectively degenerated to normal *XPath*, by ignoring point nodes and replacing ranges with all nodes that have some intersection with the range, necessary to retrieve an *XPath* node-set. A further difference introduced is that selected element nodes are replaced by their complete sub-tree resulting in a node-set that is the union of them all and may result in a forest.
The *XPointer* Implementation Report [37] mentions that the principal extensions beyond *XPath* were rarely implemented. This means that the *XPath*-only functionality of *XPointer* and the `xpointer()` scheme [35] could easily be propagated to recommendation and formally and normatively supported for such a subset. *XSECT* for instance provides support up to this level.
There are *XPath*-only schemes defined for the *XPointer* framework in the XPointer Scheme Registry which however appear to be hardly noted or supported. A reason may be that such a registry does not have the normative standing of a recommendation and is hence not as prominent.

Why has the `xpointer()` scheme not been removed in *XMLDSIG SE*?

Although there is weak commitment towards *XPointer* in general and in *XMLDSIG SE*, the XML Security Specifications Maintenance Working Group (*XSSMWG*) discovered that the *XPointer* candidate recommendation was normatively referenced by *XMLDSIG* and has to remain so for conformance in *XMLDSIG SE*. This had to be done despite the fact that the `xpointer()` scheme has no normative standing of a *W3C* recommendation[15] , because there has been a period of almost seven years where this has not been uncovered.

Why has the `xpointer()` scheme never become a recommendation?

Reasons for not having made it to a recommendation until today are rooted in US patent claims potentially affecting parts of the *XPointer* specification. Nevertheless this has not been uncovered in the *XMLDSIG* community for a long time, and so several specifications and implementations based on *XMLDSIG* used the *XPointer* candidate recommendation. There has been considerable controversy [38] and comments mentioning prior art. The situation is less drastic in Europe as in principle software as such is not patentable. Uncertainty concerning a first license, which has been superseded by a second license did obviously not convince the community surrounding *XPointer* until today to provide enough complete implementations of the `xpointer()` scheme to move it to a recommendation.
Despite the normative difficulties, a subset of *XPointer's* syntax has been used and is mentioned in examples of the *XPointer* framework [30]. The Austrian Citizen Card makes use of the `xpointer()` scheme in its Security Layer [39][16] as well as German banking associations in Electronic Banking Internet Communication Standard (*EBICS*).

Do *XPointers* have to be escaped in *XMLDSIG*?

URI references in *XML* can use characters that are not allowed in *URIs* is general. This has however been specified across many specifications and in the case of *XMLDSIG* lead to some ambiguities. One might think the following full *XPointer* in expression 2.3.22 taken from the *EBICS* standard can be used as it is in *URIs* in *XMLDSIG*.

```
<Reference URI="#xpointer(//*[@authenticate='true'])">          (2.3.22)
```

There are a few subtleties with the use of *URIs* in *XMLDSIG* in combination with *XPointer*. It is unclear whether a fragment-only *URI reference* containing “unescaped square brackets `[]`” as in expression 2.3.22 is allowed in *XMLDSIG* or whether it has to be percent-encoded as in expression 2.3.23.

```
<Reference URI="#xpointer(//*%5B@authenticate='true'%5D)">          (2.3.23)
```

The grammar in RFC2732 [28] (Figure 2.3.25) as opposed to its prose allows them.
The following characters “`a..zA..Z0..9-._~!$&'()*+,;=/?:@`” are allowed in a fragment by the grammar in RFC2396 [26] (Figure 2.3.24).
RFC2732 [28] moves the square brackets `"["` and `"]"` to the list of reserved characters and hence allows “`a..zA..Z0..9-._~!$&'()*+,;=/?:@[]`” in the fragment.

[15]Standardization within the *W3C* follows a certain process before becoming a *W3C* Recommendation. A process graphic is shown here `http://www.w3.org/2001/02pd/rec54-img.svg`.

[16]Examples can be found in their tutorial.

```
fragment       = *uric
uric           = reserved | unreserved | escaped
reserved       = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                 "$" | ","
unreserved     = alphanum | mark
mark           = "-" | "_" | "." | "!" | "~" | "*" | "'" |
                 "(" | ")"

fragment       = *(
                  ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                  "$" | ","
                   alphanum |
                  "-" | "_" | "." | "!" | "~" | "*" | "'" |
                  "(" | ")"
                   )
```

Figure 2.3.24: RFC2396 [26] *BNF* relevant for *URI* fragment

```
   (3) Add "[" and "]" to the set of 'reserved' characters:

      reserved    = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                    "$" | "," | "[" | "]"

   and remove them from the 'unwise' set:

      unwise      = "{" | "}" | "|" | "\" | "^" | "`"
```

Figure 2.3.25: RFC2732 [28] *BNF* relevant for *URI* fragment

XMLDSIG seems to allow `"["` and `"]"` in a fragment, but the intent expressed in the text of RFC2732 [28] is that they should be used only for IPv6 hosts and not in the fragment.

> It defines a syntax for IPv6 addresses and allows the use of `"["` and `"]"` within a URI explicitly for this reserved purpose. [28]

Hence the only solid choice is to escape square bracket characters unless they delimit an IPv6 host. *XMLDSIG* also appears to allow `"#"` and `"%"` (section 4.3.3.1 [27]), but again the only solid choice is to escape these characters unless they delimit the fragment from the path or start an percent encoding respectively. *XMLDSIG SE* clarifies the situation (section 4.3.3.1 [29]) in accordance with the `xs:anyURI` *Schema* type. It references now RFC3986 [25] and it is likely that many *XML* related *W3C* recommendations will in future versions at least non-normatively refer to Legacy extended IRIs for XML resource identifications (*LEIRI*) that can be mapped to Internationalized Resource Identifierss (*IRI*) and eventually to *URIs*.
The addition of unescaped square brackets to be allowed in the fragment would have complicated the very simple escaping algorithm of *LEIRIs* and the weak standing of the `xpointer()` scheme made such an addition unjustifiable and it was hence not adopted by *XMLCORE*.

2.3.6 Extensible Stylesheet Language for Transformation (*XSLT*)

The Extensible Stylesheet Language for Transformation (*XSLT*) is used to transform *XML* data and documents to *XML* and other data formats. *XSLT* is located in the Transformation API for XML (*TrAX*) of *JAXP*. The target data format is often *HTML*, plain text or *XML* based outputs like *XHTML* or *PDF* via Extensible Stylesheet Language Formatting Objects (*XSL-FO*). *XPath*, *XSLT* and *XSL-FO* are referred to as Extensible Stylesheet Language (*XSL*) technologies.
There are different approaches how *XSL* style-sheets can be written. This section very briefly discuss how to write a style-sheet transformation. *XSLT* in its most frequently used variant is data driven and defines templates that match nodes of an input node-set by means of an *XPath* expression as soon as they are passed in. Such templates then produce the output by printing their child elements and content from other namespaces than the `xsl` [17] namespace. These are further combined and intermingled with values and nodes of the matched node-set from the input document. These style-sheets often work with built-in template rules (see Figure 2.3.26) and automatically process an input document by recursion.

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="comment()|processing-instruction()"/>
```

Figure 2.3.26: Built-in Template Rules

- The first built-in template in Figure 2.3.26 matches the root of the document `"/"` and any element node `"*"` including the document element.

  Then `xsl:apply-templates` is performed which in turn defaults to selecting all child nodes by `select="*"` (same as `select="child::*"`) and applying the matched template. This is hence forming an automatic depth-first recursion over the elements of an input document.

- The second template causes all text nodes and attribute values to be printed as they are discovered in the node-set. Note however that the first built-in template does not iterate over the attributes.

- The third built-in template causes *comments* and *processing instructions* to be ignored by default. It has no child nodes that either apply or call other templates and does not contain nodes that could be emitted. Empty templates are essentially no-ops and if matched just consume nodes.

[17]Recall that a prefix can be associated with a namespace. In this case `xsl` would be declared like this: “`xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`”

To override the second and third built-in template comprises a common approach to write a style-sheet and may be combined with additional templates that match node-sets according to the value of `match`. This approach is similar to pattern matching in functional programming languages and takes advantage of the built-in recursive depth-first traversal.
Another approach is to override the first built-in template and to explicitly select and iterate the input nodes by using commands like `<xsl:for-each>` potentially in combination with named templates that are similar to subroutines. Such an approach pulls the nodes in and calls certain templates with certain node-sets, whereas the first approach is pushed by recursions. The pull approach is closer to an imperative way of processing the data. Also a hybrid approach may be taken by calling `<xsl:apply-templates>` again on sub-trees of the document.

XSLT data types

XSLT supports commonly used data types like number, string, boolean and data types more specific to *XML* like node-sets or the rarely used result tree fragments. Result tree fragments are the type of variables that contain a number of nodes from the actual stylesheet itself and hence can be emitted as output. Selecting nodes or only parts of a variable of type result tree fragment is usually not possible in *XPath* 1.0 as they are not considered to be node-sets. Many implementations however offer proprietary extensions to convert result tree fragments to node-sets.

Important language constructs

Language constructs include conditionals like `<xsl:if>` which however misses an “else” path. This is mitigated by `<xsl:choose>` that can contain one or more `<xsl:when>` conditionals and one `<xsl:otherwise>` construct. Node-sets in *XSLT* can be iterated by using `<xsl:for-each>` and using the `position()` for getting an index.
As mentioned above there are two approaches to actually execute templates, the one is done by pattern matching using `<xsl:apply-templates>` the other one `<xsl:call-template>` calling a pattern by its name. Different templates matching the same nodes are valid and templates can carry the same name if distinguished by a mode attribute. The mode attribute is like a string parameter specified with `<xsl:apply-templates mode="mode1"/>` or `<xsl:call-template name="foo" mode="mode2"/>`.

Template priorities, import and include

Default priorities modeled into the language exist trying to match the most specific pattern when determining the template to be called.
Style-sheets may also import or include other style-sheets. Templates from imported style-sheets have a lower priority than the “local” templates of the importing stylesheet.

When a stylesheet however is included, the included templates receive the same priority classification as if they would have been declared directly in the including style-sheet.

Extension mechanism

XSLT has an extension mechanism and allows for so called extension elements and extension functions, that can pose a significant security risk [40] (see also. subsection 4.2.4) and are usually implementation dependent. At the time of writing version 2.0 became a *W3C* Recommendation. It does not play a role for the current version of *XMLDSIG* or *OASIS-DSS*. *XSLT* is part of *TrAX* the Transformation *API* for *XML*.

2.3.7 Cascading Style Sheets (*CSS*)

Cascading Style Sheets (*CSS*) define the appearance of *XML* or *HTML* documents separately form the actual data. They are usually included by means of a *processing instruction*. *CSS* is a simple declarative language[18] that allows authors and readers to attach styles such as fonts, colors and spacing to *HTML* or *XML* documents. Authors can tailor their documents for presentation on various media such as visual browsers, aural devices, printers, Braille devices, and hand held devices [11]. “Stylesheets affect security because they must be included under a signature if the signature is meant to securely indicate approval of information as presented to a user.” [11]

2.4 Digital Signatures

Digital signatures, or more generally electronic signatures strive to be the electronic counterpart of handwritten signatures. Digital signatures in the context of this thesis are clearly distinct from handwritten electronic signatures performed on a touch sensitive pad or display. The latter signatures today often appear at cash registers in US or UK supermarkets or shops and replace paper credit card payment instructions. In continental Europe personal digital assistants (PDA) are often used as a replacement for a paper signature to acknowledge the receipt of a parcel from the postal service.
In contrast digital signatures employ a cryptographic signature scheme such as RSASSA-PKCS 1-v1_5 specified in Public Key Cryptography Standards (*PKCS*) #1 or Keyed-Hash Message Authentication Code (*HMAC*) to assure authenticity, integrity and potentially non-repudiation of signed data. It should be noted that digital signatures, in contrast to electronic handwritten signatures, do not require physical presence of the signatory. It is hence possible to sign data in a client server environment for which *OASIS-DSS* can be the interface.
In [8] the relation of handwritten signatures to digital signatures is discussed and short introductions to Public-Key Infrastructures (*PKI*), public signature schemes, qualified certificates and revocation of

[18]*CSS* itself is not using an *XML* syntax, because its early versions predate *XML*. Newer versions of *CSS* may not have switched to an *XML* syntax because Lie did not want to end up with a programming language [41].

certificates are given. The source points out that all known techniques that meet the requirements for secure electronic signatures laid out by European regulators are based on cryptographic methods and mentions further that handwritten signatures written on a touch sensitive display are not secure electronic signatures. Given the source is a little dated with respect to its key length recommendation, we add that today as of 2008 the recommended key length for the RSA signature scheme has increased from 1024 Bit to 2048 Bit [42]. Lenstra and Verheul projected in 2001 that a machine build in 2009 for $250 million could factor a 1024-bit RSA key in a day. A good resource is `http://www.keylength.com/` and provides a good overview of key length recommendations.

It should further be added that longer hash functions of the SHA-2 family should be considered for future systems. In [43] we discuss the need to move from SHA-1 to longer hash functions based on FIPS and NIST draft publications and recent advances in search for SHA-1 collisions by Rijmen, Mendel, Rechberger, De Cannière [44], [45].

XML Advanced Electronic Signatures (*XAdES*) [46] and CMS Advanced Electronic Signatures (*CAdES*) [47] argue for the need for certificate revocation when a private key is lost and the implied importance of timestamps to provide lower (former) and upper (later) bounds for the actual signature creation time and [8] gives a good overview about this.

It should be added that the lower bound and upper bound are considered important by some legislation within Europe to record the time of creation of the signed data by augmenting it with a time-stamp.

This allows to interpret claimed times and dates inside the signed data in the light of these bounds. The upper bound additionally prevents repudiation by certificate revocation after the upper bound in time. In practice these two bounds should ideally form a small interval, that in good approximation could be seen as an instant in time. However, timestamps are in most cases not required to make valid signatures despite in some special cases like patent applications or similar.

Cryptographic signature schemes provide the basis for creating secure electronic signatures and advanced electronic signatures compliant with European legislation.

In Europe electronic signatures are treated in Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a "Community framework for electronic signatures".

An introduction to the European legal framework for electronic signatures can be acquired from various sources (e.g. [48] section 1.1).

2.4.1 XML Digital Signatures

XMLDSIG is a *W3C* recommendation [27] and The Internet Engineering Task Force (*IETF*) draft standard revised by the *XSSMWG*. Eastlake [11] gives an introduction to *XMLDSIG* and an overview can be found in section 2.0 of *XMLDSIG* [27][29].

XMLDSIG's structures are defined by means of a *DTD* and a *Schema*. The *Schema* follows primarily the Garden of Eden *Schema*-pattern introduced in subsection 2.2.2.3. There are a few exceptions, i.e. the element declarations for <ds:HMACOutputLength>[19], <ds:XPath>, all element declarations of type ds:CryptoBinary and all child structures of <ds:X509Data> are defined locally.

Figure 2.4.1 [27] shows the structure of *XML* signatures using *BNF* like quantifiers like `?` for zero or one occurrence, `+` for one or more occurrences and `*` for zero or more occurrences of an element in a given location of the structure. To emphasize the fact that elements are in the *XMLDSIG* namespace the prefix `ds:` has been added to Figure 2.4.1.

```
<ds:Signature ID?>
  <ds:SignedInfo ID?>
    <ds:CanonicalizationMethod/>
    <ds:SignatureMethod/>
    (<ds:Reference URI? ID?>
      (<ds:Transforms>)?
      <ds:DigestMethod>
      <ds:DigestValue>
    </ds:Reference>)+
  </ds:SignedInfo>
  <ds:SignatureValue ID?>
  (<ds:KeyInfo ID?>)?
  (<ds:Object ID?>)*
</ds:Signature>
```

Figure 2.4.1: XML Digital Signatures Overview [27] with minor update.

A top level overview on how *XML* documents are signed and verified is given in the following Data Flow Diagram (*DFD*) (Figure 2.4.2). The *DFD* notation has been extended by a circle with a double lined border to represent the document that will eventually bear the signature. This document may well be in a location addressable by an *URI* (Figure 2.4.2 3.). Further a circle with a thick border was added to represent an information drain. The *DFD* should be read so that the data takes one path along directed edges until resulting in a <ds:DigestValue>, signed document or verification result.

Figure 2.4.2 describes the process of signing and employs an *XML* digital signatures library (1. XMLDSig) that builds upon a library that offers cryptographic functionality (CryptoLib) and potentially upon a hardware security module (HSM) allowing better protection of private key material. The signer (Signer) provides via some *API* for *XMLDSIG* a set of inputs to create <ds:Reference> elements (Reference+) identifying, filtering and transforming data objects to be signed. If a Java platform is used this *API* is usually *JSR105*. Further an *URI* specifying how the set of <ds:Reference> elements comprising the <ds:SignedInfo> will be canonicalized (<ds:CanonicalizationMethod>) and a set of inputs to create a <ds:KeyInfo> providing information about the signer and how to discover the `VerificationKey` (KeyInfo) is given. The optional <ds:KeyInfo> may in the case of a public signature scheme bear the signers public key. This Information (KeyInfo) is if available also used to identify and retrieve the SigningKey.

[19]Note that elements will be shown as "<prefix:localname>", types as " prefix:localname " and attributes as `Attribute`, use also page viii.

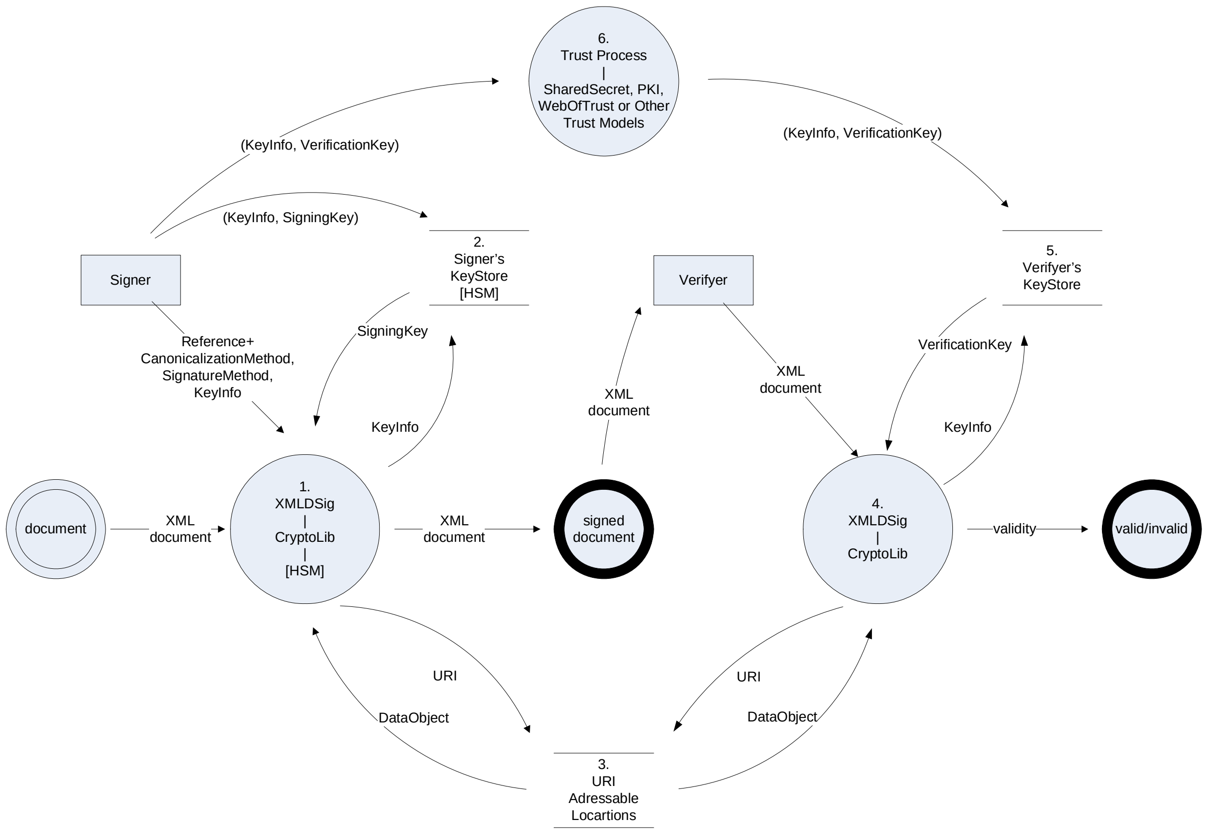


Figure 2.4.2: XMLDSIG DFD Level 1

A verifier then receives the signed document and again uses an *XML* digital signatures library (4. XMLDSig) and the embedded KeyInfo to retrieve the VerificationKey[20]. The references are processed to generate the <ds:SignedInfo> which is then canonicalized, digested and the resulting digest value is compared against the digest value secured in the <ds:SignatureValue>. This is called the core processing of *XMLDSIG*.

2.4.1.1 Processing Model

The introduction to the processing model given here approaches *XML* digital signatures by taking a look at the data flow of the signed data objects.

These are referred in the URI attribute of <ds:Reference> (Figure 2.4.1 line 5), filtered[21] or transformed by the <ds:Transforms> (line 6) and finally resulting in the hash (<ds:DigestValue> line 8). The data flow of the hash values is continued and they are collected in the <ds:SignedInfo> (lines

[20]As mentioned in chapter 1 we prefer to talk about verification of signatures in contrast to *JSR105* speaking about a `validatingKey` and using a method called validate to verify a signature.

[21]It is easier to understand many of *XMLDSIG's* idiosyncrasies when *XPath*-filtering and *XPath*-Filter 2.0 [49], although provided in a <ds:Transform>, are not viewed as transforms. They are filters or selectors and depend not only on the input node-set but also on the whole input document.

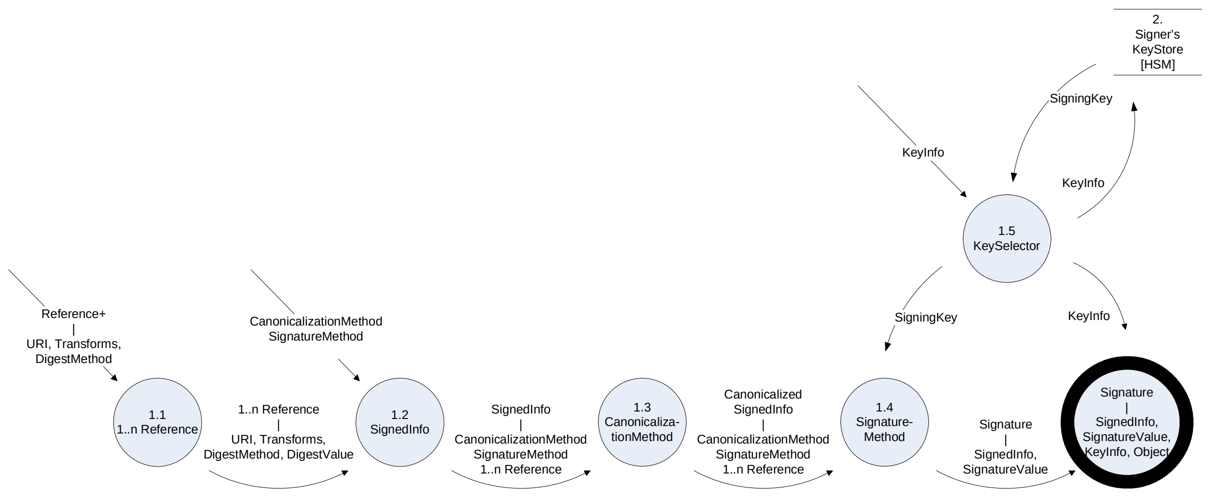

2. Signer's KeyStore [HSM]
SigningKey
KeyInfo
KeyInfo
1.5 KeySelector
SigningKey
KeyInfo
Reference+ | URI, Transforms, DigestMethod
CanonicalizationMethod SignatureMethod
1.1 1..n Reference
1..n Reference | URI, Transforms, DigestMethod, DigestValue
1.2 SignedInfo
SignedInfo | CanonicalizationMethod SignatureMethod 1..n Reference
1.3 Canonicaliza-tionMethod
Canonicalized SignedInfo | CanonicalizationMethod SignatureMethod 1..n Reference
1.4 Signature-Method
Signature | SignedInfo, SignatureValue
Signature | SignedInfo, SignatureValue, KeyInfo, Object

Figure 2.4.3: XMLDSIG DFD Level 2: The Processing Model (sign)

2-10) which is normalized by application of the <ds:CanonicalizationMethod> (line 3, aka. canonicalized, see section 2.5) and produces, by application of some signature scheme identified by the <ds:SignatureMethod> (line 4) a <ds:SignatureValue>.
To assure a secure binding of the signer to its VerificationKey, some sort of trust process building on a trust model is required. In the simplest way this is a shared secret used for *HMAC* or for public key cryptography some *PKI* where a trusted path to a trusted third party can be established.
How <ds:Reference>s are processed is described first. They comprise the <ds:SignedInfo> which is then canonicalized, digested and signed eventually resulting in the <ds:Signature>. This order is somehow given naturally by the need to have the digest values before they can be complied to a <ds:SignedInfo> on signing, for verifying the reverse order is possible and preferable [40] because <ds:Transforms> usually come form a foreign source and may be a security problem (recall section 2.3.6 on page 35 and more discussion on this will be in subsection 4.2.4).
XMLDSIG has a reference mechanism, in contrast to other signature formats like Cryptographic Message Syntax (*CMS*) that only sign one data object. This reference mechanism employs one *URI*[22] per <ds:Reference> to identify the data objects located either within the same *XML* document as the signature or anywhere else accessible via an *URI*. Certain forms of signatures can be characterized by the way the data objects are positioned in relation to the signature in the same document or externally. So before continuing on the flow of data objects, an overview of the different forms of referring to the data objects is provided.

2.4.1.2 Signature Forms

- Enveloped Signatures are characterized by the fact that the signature is a descendant of a part of the signed data object.

[22]One reference may omit the URI in which case dereferencing is application dependent.

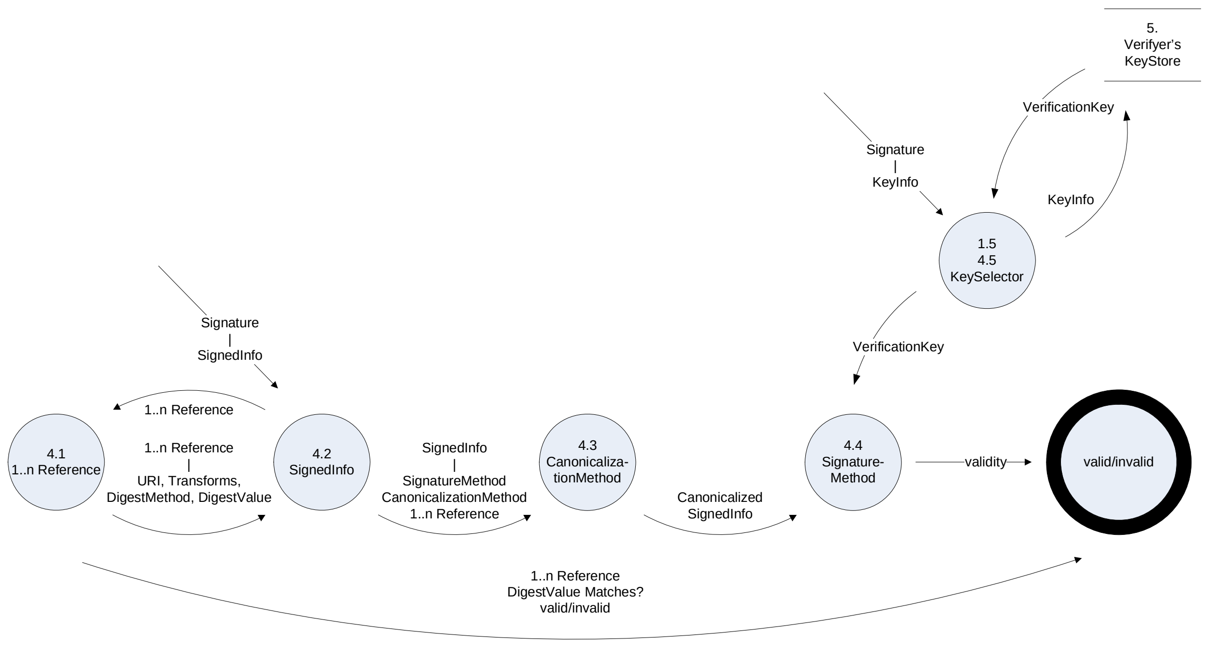


Figure 2.4.4: XMLDSIG DFD Level 2: The Processing Model (verify)

Enveloped signatures are placed inside the signed data object and hence one of the signature's ancestors is actually dereferenced by a reference within the signature itself. Enveloped signatures have to exclude themselves[23] from the data that is signed. This is performed by the so called enveloped signature transform which is based on the `here()` function. The `here()` function is an extension to *XPath* returning a node-set comprised of the node containing the expression.

- Enveloping Signatures bear the signed data object in the <ds:Object> and hence the signature is an ancestor of the signed data object.

Enveloping signatures take advantage of the <ds:Object> (line 13) that can be used to embed *mixed content* or just a text node representing data. *XML* documents however can have a *prolog* and hence the element content production does not match the document production. This implies that one can not envelope *XML* documents in general. Issues with the *prolog* and inherited namespaces can arise (see also section 4.1). Hence *XML* documents are like binary data objects often base64 encoded and placed inside a <ds:Object>.

- Detached Signatures are completely disjoint from the signed data object.

Detached signatures are disjoint from the signed data object and may lie within the same document as the data object or in a separate file.
When *XPointer URI* fragments or more <ds:Reference>s than one are used then combinations of these different forms with respect to the data objects can be achieved.

[23]At least the digest value of the corresponding reference and the signature value have to be excluded.

Having explained the way data objects are referred by an *URI* and how this characterizes the different forms of *XML* signatures we turn our focus back on the data flow.

2.4.1.3 Reference Processing Model

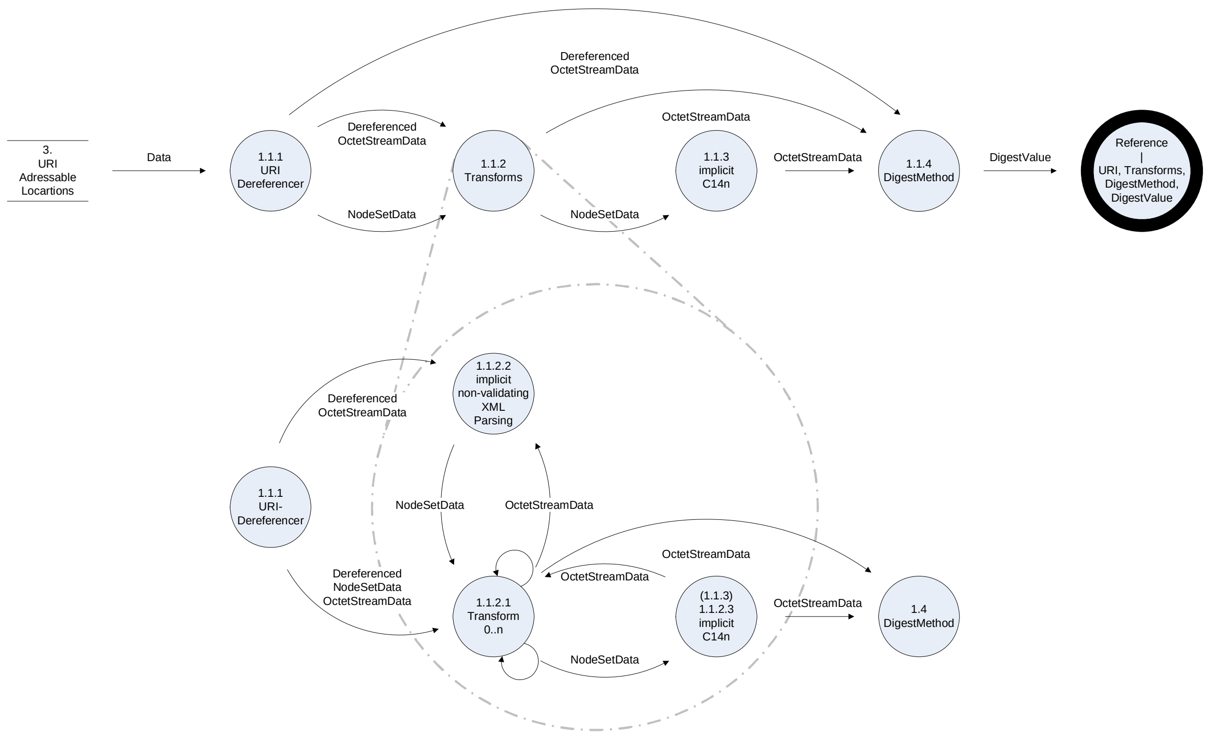


Figure 2.4.5: *XMLDSIG DFD* Level 3: The Reference Processing Model (sign)

How each reference is processed is shown in Figure 2.4.5. This processing is largely equal on signing and verification with the addition of a comparison of the <ds:DigestValue> with the existing one for verification. After the *URI* has been resolved the data object (`Data`) is retrieved via the `URIDereferencer`[24] (1.1). Depending on the *URI reference* the `URIDereferencer` returns `NodeSetData` for so called *same-document references* (wrt. *XMLDSIG* fragment only *URI references* Figure 2.3.17) and refer to the whole or portions of the same document as the <ds:Reference> in question. `OctetStreamData` is returned for external *URI references*.
If `NodeSetData` was dereferenced it should be noted that depending on the form of the fragment only *URI* the node-set will contain *comments* for full *XPointer* expressions but not for shorthand *XPointer* expressions. *XMLDSIG* however does not clarify whether *comments* are considered to be removed at parse time or whether they are considered to be in the data model (subsection 4.2.8).
The dereferenced data will be filtered and transformed by the <ds:Transforms> (1.2).

[24]The terms `URIDereferencer`, `Data`, `OctetStreamData` and `NodeSetData` correlate with class names in the *JSR105*.

A <ds:Transform> (1.2.1) can require `NodeSetData` or `OctetStreamData` as input either from the `URIDereferencer` or a preceding <ds:Transform>. The *XMLDSIG* processing model mandates that there is an implicit conversion to be performed on input from `OctetStreamData` to `NodeSetData` via implicit non-validating *XML* parsing (1.2.2) and vice versa using *C14n* (1.2.3).
The last <ds:Transform> (Transform) will either return `OctetStreamData` that is then directly hashed (1.4) or `NodeSetData` that will implicitly be canonicalized (1.3) and then hashed (1.4) resulting in a <ds:DigestValue>.

2.5 Canonicalization

Canonicalizing XML is hard! Tim Bray[25]

To be able to digest *XML* we need a binary representation or serialization, because only a series of bytes (aka. octets) can be signed. Certain aspects of *XML's* serial representation are left open and a canonical and reproducible representation is hence required.
The goal of canonicalization is to remove any information, that is considered certainly insignificant and to define an unambiguous representation for aspects that can be represented in various ways. Such negibilities range from character encoding, line breaks, order of attributes, whitespace in tags and between attributes, unutilized namespaces to value normalizations based on a *DTD* or *Schema*.
Higher forms of canonicalization include the more primitive ones.
The following forms of *XML* canonicalization currently can be found in standards, drafts and other sources. They are presented here by their level of sophistication and ordered from simple to complex:

- Minimal Canonicalization (*MC14n*) [50] [51]
- Canonical XML Version 1.0 (*C14n*) [52]
- Canonical XML Version 1.1 (*C14n11*) [53] fixing issues analyzed by us [54] and the *XMLCORE* working group (*WG*).
- Exclusive XML Canonicalization Version 1.0 (*Exc-C14n*) [55]
- Schema Centric XML Canonicalization Version 1.0 (*ScC14n*) [56]

2.5.1 Minimal Canonicalization (*MC14n*)

Eastlake briefly discusses the trade-off between insufficient canonicalization and excessive canonicalization (section 9.1 [11]). He introduces canonicalization as being useful for signing text and mentions Minimal Canonicalization (*MC14n*) (section 2.4 [51]) defined in section 6.5.1 of RFC3075 [50][26].
MC14n only performs encoding and character level normalizations:

[25]`http://lists.xml.org/archives/xml-dev/200403/msg00305.html`
[26]obsoleted by RFC3275[57] which does not contain *MC14n* any more.

- trans-code to UTF-8
- Unicode normalizations, if converted from non-Unicode
- line end (line break, Unix, Windows)

MC14n has to be used explicitly in *XMLDSIG* by using a <ds:Transform> on the <ds:Reference> level to canonicalize data objects and as <ds:CanonicalizationMethod> on a <ds:SignedInfo> level. The use of *MC14n* raises interoperability concerns, because as of 2005 two independent interoperable implementations of *MC14n* have not been announced [51].
After mentioning that canonicalization shall remove what is insignificant and maintain what is significant to an application, Eastlake lapidary points out that: "achieving robust and secure signatures requires just the right canonicalization [11]".
What is less clear from this source is that getting canonicalization right, may be highly application dependant.

2.5.2 Canonical XML Version 1.0 (*C14n*) and Canonical XML Version 1.1 (*C14n11*)

Canonical XML Version 1.0 (*C14n*) and Canonical XML Version 1.1 (*C14n11*)[27] go beyond the character level normalizations and are concerned with the normalization of logical structures of *XML*.
They can however only achieve normalization of *XML* as far as the *XML* specification [6] designates parts of its serialized representation as insignificant. The *XMLNS* specification [58] has been examined to determine that superfluous namespace declarations (section 4.6 [52]) can be removed. Such are essentially unnecessary namespace redeclarations and they have to be clearly distinguished from unused namespace declarations. The latter are namespace declarations which are not used by any of the elements or attributes that have them in scope.
C14n retains all whitespace content as parsers pass it on to applications. A *DTD* or *Schema* is required to distinguish ignorable whitespace of pure element content from whitespace in more general *mixed content*. As however *C14n* is ignorant about whether a *DTD* or *Schema* is available and an *XMLDSIG application MUST attempt to parse the octets yielding the required node-set via XML well-formed processing*, it is best common practice to retain as much white space as possible. The alternative to let the parser remove the ignorable whitespace (see also subsection 4.1.3) is prohibited by *C14n*:

> All whitespace within the root document element[*sic!*][28] MUST be preserved (except for any #xD characters deleted by line delimiter normalization)(section 2.1 of [52][53])

So *C14n* adds the following aspects to *MC14n*:

- attributes are lexicographically ordered, their values normalized and delimited by double quotes

[27]*C14n* and *C14n11*can be used interchangeably in most of this paragraph.
[28]Meant was obviously the *document element*.

- empty elements are represented unambiguously
- CDATA sections are replaced with their character content
- superfluous namespace redeclarations are removed from each element
- namespace declarations are lexicographic ordered
- *xml declaration* and *DTD* are removed
- Some normalizations required by *C14n* are preformed by parsers
  - character and parsed entity references are replaced (expanded)
  - whitespace outside of the *document element* is removed
  - whitespace within start (space between attributes) and end tags is normalized
  - default attributes are added to each element in validating parsers[29]
- some normalizations of *DTD* validating parsers need to be suppressed to assure that
  - all whitespace in character content is retained (excluding characters removed during line feed normalization)[30]

Although *C14n* has been superseded by *C14n11*, it is however still the default canonicalization for `NodeSetData` to `OctetStreamData` conversion (Figure 2.4.5 on page 41 Item 1.1.3) and is the only canonicalization that is applied implicitly. <ds:SignedInfo>'s <ds:CanonicalizationMethod> (line 3 Figure 2.4.1 on page 37) is not inherited as a default for <ds:Reference> level processing.
C14n11 as the successor of *C14n*[31] fixes problems surrounding the inheritance of attributes in the *XML namespace*. The first example in Figure 2.5.1 on page 45 shows that *C14n* erroneously passes *xml:id*, which has been defined after *C14n*, down to orphaned nodes [54], despite *xml:id* is not an inheritable attribute. Hence we introduced[32] the term *simple inheritable attribute* for *xml:lang* and *xml:space* to distinguish them from *xml:base*, which has more complex inheritance rules.
Similarly *xml:base* was not treated correctly in *C14n* [54]. The second example in Figure 2.5.1 on page 45 shows this and rules for passing on a *relative URI reference* value in *xml:base* have been specified in *C14n11* for node-sets representing fragmented document sub-trees.

[29]Requires a validating parser configured to perform http://xml.org/sax/features/validation and a *DTD* or *Schema*

[30]Requires for a validating parser on *SAX* level http://apache.org/xml/features/dom/include-ignorable-whitespace to be set to true or at the *DOM* level setIgnoringElementContentWhitespace(false) or setting the parameter element-content-whitespace

[31]For signature generation the explicit usage of *C14n11* is recommended by *XMLDSIG SE* [29], which implies that *C14n* is not recommended to be used any more for new signatures.

[32]"`http://lists.w3.org/Archives/Public/public-xml-core-wg/2006Mar/0040.html`"

input	*C14n*-output	*C14n11*-output
XPath document subset expression		
`(//. | //@* | //namespace::*)[ancestor::a and not(. = parent::a/@*)]`		
`<a xml:id="ida">`		
`  <b />`	`<b xml:id="ida"/></b>`	`<b></b>`
`  <c />`	`<c xml:id="ida"/></c>`	`<c></c>`
`</a>`		
XPath document subset expression		
`(//. | //@* | //namespace::*)[not(self::b or . = parent::b/@*)]`		
`<a xml:base="p/f">`	`<a xml:base="p/f">`	`<a xml:base="p/f">`
` <b xml:base="..">`		
`   <c xml:base="x">`	`   <c xml:base="x">`	`   <c xml:base="../x">`
`   </c>`	`   </c>`	`   </c>`
` </b>`		
`</a>`	`</a>`	`</a>`

Figure 2.5.1: Example outputs for *C14n* and *C14n11*

2.5.3 Exclusive XML Canonicalization Version 1.0 (*Exc-C14n*)

Exclusive XML Canonicalization Version 1.0 (*Exc-C14n*) avoids to fix up any inheritance of simple inheritable attributes and *xml:base*, by simply treating them as normal attributes. This is fine for connected (non-fragmented) node-sets, but may cause problems in fragmented ones as we will see in subsection 2.5.3.1. *Exc-C14n* [55] adds to the notion of superfluous namespace declarations the notion of *visibly utilized namespace declarations*. Such can be understood as the complement of unused namespace declarations. The term becomes more important where *Exc-C14n* modifies the processing of *C14n* for namespace declarations. *Exc-C14n* only renders namespace declarations for sub-trees of an *XML* document (or fragmented parts thereof), that really use the associated prefix in elements or attributes. *Exc-C14n* modifies the processing of *C14n* as follows:

- simple inheritable attributes and *xml:base* are treated like normal attributes
- an optional parameter called `InclusiveNamespacePrefixList` containing a list of space separated namespace prefixes that will behave as in *C14n* can be supplied
- all other namespace prefixes will be rendered only where needed in element or attribute names and unused namespace declarations are hence removed.

It should be added that namespace declarations solely needed for *QNames* in content - like in *XPath*-expressions that use *prefixed names* - are not considered as being *visibly utilized namespace declarations* and should hence be added to the `InclusiveNamespacePrefixList`. This may seem surprising as this is a well known issue. Eastlake mentions that namespace prefixes are considered significant in such *XPath*-expressions (section 9.1 [11]) and they can appear inside attribute values or text content.

A reason may be that Eastlake considers it *generally impossible to determine algorithmically whether a namespace prefix is actually referenced by some XML [11].* Eastlake points to some problems with *Exc-C14n* (Section 9.6.4 [11]), but provides little guidance.
It is good practice to use *Exc-C14n* only for connected node-sets and declare all used prefixes in the `InclusiveNamespacePrefixList`.
The following section will hence discuss how to determine what data is suitable for *Exc-C14n*.

2.5.3.1 `NodeSetData` suitable for exclusive canonicalization

In general it is good practice to use *Exc-C14n* whenever possible, especially if applications use namespace prefixes only to qualify elements and attributes whose owning element is also in the document subset. Despite the fact that document sub-sets (node-sets) containing attributes and not their owning elements have a questionable semantic and hence should be avoided, they are nonetheless allowed in *XPath* and accepted by *Exc-C14n*. Such node-sets are however not suitable for *Exc-C14n* with respect to the definition of *visibly utilized namespace declarations*.
If no default namespace has been specified there will be only *prefixed names* and names that are in no namespace. If further namespace prefixes are used only in attributes, whose owning element is in the document subset and in element names then the *visibly utilized namespace declarations* will be equal to the used namespace declarations.
This could even be checked programmatically by a heuristic that finds *prefixed names* by matching the production in attribute values and text nodes. If however a default namespace has been declared *QNames* do not necessarily have a prefix and can hence not be found easily.
Hence the default namespace should be added to the `InclusiveNamespacePrefixList` in any case. Adding `#default` will assure the correct interpretation of *QNames* without prefix.
The `InclusiveNamespacePrefixList` allows to sign *XML* using *Exc-C14n* in cases where all prefixes that are used in *prefixed names* in content are known.
Prefixes that have been used anyway, by an output ancestor element or an output attribute of an output ancestor element of the node bearing the *QName* in question, could theoretically be exempted. In practice however it is advisable not to exempt them and to add all such prefixes as it does not harm to treat *visibly utilized namespace declarations* inclusively.
Data objects should not depend on inheritable attributes (*xml:lang*, *xml:space* and *xml:base*) unless these attributes and their owning elements are in the document subset.
In fragmented document subsets additional caution is required as orphaned elements will not necessarily inherit from their parent but rather from their grandparent or further ancestors. Figure 2.5.2 on page 47 shows this in its first example. With respect to *xml:base Exc-C14n* is also not useful in fragmented node-sets, as can be seen in the second example.
To see what content is suitable for *Exc-C14n* it is best to look at what documents including their actual information content would be destroyed if canonicalized exclusively.
The direction in which mathematical expressions are read matters and can depend on the direction, that

input	*Exc-C14n*-output	*C14n11*-output
XPath document subset expression `(//. | //@* | //namespace::*)[not(self::b or .=parent::b/@*)]`		
`<a xml:lang="de"> <b xml:lang="en"> <c /> </b> </a>`	`<a xml:lang="de"> <c></c> </a>`	`<a xml:lang="de"> <c xml:lang="en"></c> </a>`
XPath document subset expression `(//. | //@* | //namespace::*)[not(self::b || parent::b/@*)]`		
`<a xml:base="p/f"> <b xml:base=".."> <c xml:base="x"> </c> </b> </a>`	`<a xml:base="p/f"> <c xml:base="x"> </c> </a>`	`<a xml:base="p/f"> <c xml:base="../x"> </c> </a>`

Figure 2.5.2: Example outputs from *Exc-C14n* and *C14n11*

can change with *xml:lang* and a value can hence change from infinity to zero 0/x = 0 and x/0 = infinity or vice versa.

2.5.4 Schema Centric XML Canonicalization Version 1.0 (*ScC14n*)

Schema Centric XML Canonicalization Version 1.0 (*ScC14n*) [56] was specified by the *OASIS* UDDI *TC* and is even richer in normalizing the content of an *XML* document.
ScC14n is an extremely complex form of canonicalization which however comes very close to actually producing equal serializations for logically equivalent *XML*. In contrast to other forms of canonicalization it requires full schema assessment and some human engagement in the form of annotating the employed *Schema*. It is based on an augmented version of the XML Information Set (*Infoset*) as input and as data-model. It also allows for a node-set or octet-stream as input and defines how they shall be converted to an *Infoset*. Besides the UDDI context however, *ScC14n* seems to be only used in MPEG-21. Otherwise - to our knowledge - it has hardly been used, nor yet as of 2008 been implemented by major vendors of *XMLDSIG* implementations.
The *ScC14n* specification mentions a number of limitations other canonicalizations suffer in its section 1.1 [56]. The following paragraphs comment some of those limitations and mark them with a bullet point followed by the actual commentary.

- In the first limitation it is claimed that *C14n* and *Exc-C14n* become *less and less useful to practical applications of XML*, because of the *advent of Schema* and the *weak expressiveness of DTDs*.

This claim however does not seem to be justified and it is unclear whether it is true after all. Thus it can be ignored.

- *C14n* (and all previously mentioned forms of Canonicalization) do not normalize namespace prefixes.

ScC14n rightfully claims that this is in contrary to the intent of *XMLNS*.

> Note that the prefix functions only as a placeholder for a namespace name. Applications should use the namespace name, not the prefix, in constructing names whose scope extends beyond the containing document. (*XMLNS* section 4 [58])

C14n justifies not normalizing namespace prefixes by potential dependencies of *XPath*-expressions in the content (section 4.4 [52], resp. [53]). *QNames* in content - as mentioned before - are known to suffer the deficiency of not necessarily being detected as such. As mentioned earlier the *XPointer* `xmlns()` scheme [33] shows that a namespace binding for a prefix is not necessarily determined by a namespace declaration.

> It is an architectural principle of URIs that they be context-independent. It follows that the QNames that appear in an XPointer must not refer to in-scope namespaces as this would make transcription impossible in the general case.[59]

This would however require a canonicalization to either assume or know that the `xmlns()` scheme has not been used or to be able to process it in a canonical manner or to ignore *XPointers* at all. In contrast to a Technical Architecture Group (*TAG*) finding [59] referring to some larger, more global redesign of, for example, XML to address these issues, *ScC14n* tries to tackle them by Namespace Prefix Desensitization.

- *ScC14n* claims that esoteric node-sets comprised of just (one) attribute(s) constitute a minor security hole to *C14n* as they will have lost their namespace context.

Exc-C14n mentioned this in its section 5.2 [55] already, but does not fix it. Despite mentioning this, *ScC14n* does not disallow esoteric node-sets and does not require its output to be well-formed (section 2.5).

- *ScC14n* eventually identifies many of the liberties (insignificant portions) within *XML* when used with *XMLNS* and *Schema*, that are not properly covered by other canonicalizations. These range from known data-type canonicalization issues which appear to ones have been overlooked by *Schema* data-types.

***ScC14n* extends *Exc-C14n*.**

ScC14n extends *Exc-C14n* about the following normalizations:

- model group `xs:all` reordering defined in *ScC14n's* serialize method in section 3.5.1, specifically at clause 2.b.iv.3 .

- removal of insignificant whitespace in element-only (non-mixed) content defined in *ScC14n's* serialize method in section 3.5.1 specifically at clause 2.b.iv.4 (cf. *Schema* [20] section 3.4.4 clause 2.3).
- Namespace Prefix Desensitization
  - *ScC14n* allows to augment *Schema* declarations of elements and attributes with special attributes in annotations in order to associate a *URI* identifying a language such as *XPath* or *QNames* that use prefixes. The types for attributes and elements are hence augmented so that instances thereof contain values interpreted in a language known to canonicalization wrt. prefixes. The association is performed by attributes in annotations to schema definitions called `embeddedLang` or `embeddedLangAttribute`. The first associates a *URI* identifying a language statically with the type (schema definition); the latter by identifying an attribute of the annotated element schema definition whose value dynamically specifies the language.
  - *ScC14n* allows for such an association also to be done if not specified as before by fiat in some specification. *ScC14n* does it so for the type of the element named "XPath" contained in elements of type ds:TransformType and the types of "xpath" attributes in *Schema*, by augmenting them with the *URI* for *XPath*.
- namespace attribute normalization
- data-type canonicalization
  - capitalization (case-insensitivity) of Unicode characters can be locale-specific and context-dependent, mapping of case can change the length of a character sequence. Upper and lower cases are not precise duals [56].

Namespace Prefix Desensitization

Namespace prefix desensitization can be influenced "*. . . by fiat in some specification . . .*" [56], but *ScC14n* does not provide for a means to specify in some parameters which specifications have been taken into account. So over time and across different application areas implementations would interpret such a clause in different ways, that in turn would harm interoperability.

ScC14n **is hardly used.**

ScC14n has not achieved wide support, because of the following reasons:
It has been specified outside the *W3C* and may be perceived as being specific to the UDDI context, it is too costly in terms of processing as it always requires full *Schema* assessment. It cannot be parametrized to to turn off expensive normalizations, that many applications do not need.

2.5.5 Canonicalizations overview

Complexity	Robustness	Affected Layers	Canonicalization
complex ⟵ simple	robust ⟶ fragile	encoding and characters	*MC14n*
		the previous and *XML*, *XMLNS*	*C14n*
			C14n11
			Exc-C14n
		the previous and *Schema*	*ScC14n*

Figure 2.5.3: Canonicalizations overview

Chapter 3

OASIS Digital Signature Services (*OASIS-DSS*)

The OASIS Digital Signature Services (*OASIS-DSS*) specifications are *XML*-based and describe two request and response protocols by means of *Schema*. The first one is a protocol to apply digital signatures and the second one to verify them. Documents can be sent to an *OASIS-DSS* server and signatures on the documents are sent back. On the other hand documents with a signature on them, around them and next to them are sent to a server, and an answer whether the signature can be verified against supplied document and data objects is sent back [60].

3.1 Incentives for using *OASIS-DSS*

Machines in a normal office environment are often in an insecure state. It can be quite complicated to keep all systems in a well maintained state, given the different threats ranging from root-kits to viruses. Malware is carried via various attack vectors such as private email and USB sticks. A protocol like *OASIS-DSS* can help to avoid to have corporate keys on such systems.
It also enables authenticated clients to sign corporate documents - like press releases - centrally using a common key. Signatures can be created and verified without complex client software and decentralized configuration.
Centralized access control, auditing and archiving of sign and verify requests can be accomplished more easily. A central enforcement including certain bylaws, policies and documents or check for certain business logic on signature generation and verification may be an advantage. Also checking of certificates by means of Online Certificate Status Protocol (*OCSP*) or Certificate Revocation Lists (*CRL*)

can be managed easier. Certificate path validation can be performed centrally and the burden is shifted away from clients.

Another important property of signing and verifying documents using an *OASIS-DSS* request is: Documents do not actually have to be in the locations indicated by the Uniform Resource Identifiers (*URI*). So signatures may still be verified, although some data object has been removed from its original location or is temporally not available. Arbitrary data can be associated with arbitrary *URI references* in *OASIS-DSS* requests. So the actual data retrieval is detached from the process of signing or verifying. Hence data stored during signing can be supplied for later verification, which is in particular useful, if web servers ceased to exist or changed their contents. An *OASIS-DSS* sign-request may even be kept as an archive format for data objects and later transformed into a verify request for signature verification. The alternative, to reestablish the environment that existed during signing, is often neither useful nor possible.

Note: An *XMLDSIG* signature as such cannot provide any assurances on the retrieval actions themselves. It does not provide evidence about the real storage location of external data objects. A successfully verified *XMLDSIG* signature merely states that the signer's and the verifier's environment retrieved the same data objects according to the *URI references* in the signature.

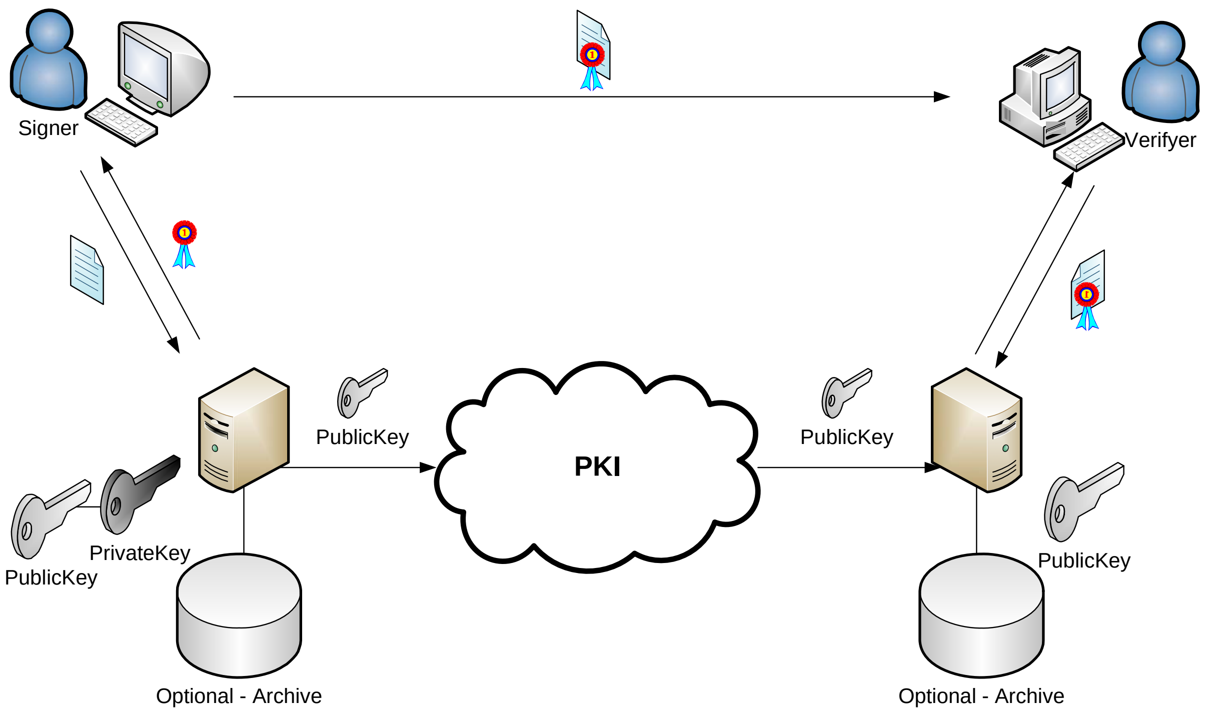


Figure 3.1.1: Signature processing with *OASIS-DSS*

Note: The following sections will contain parts of the *OASIS-DSS* specification and the schema design will be criticized - where appropriate. As mentioned in chapter 1 the work on this thesis started when *OASIS-DSS* was a working draft in revision 30 (WD30). It was neither possible nor the intention of the author's participation to conduct a complete redesign of the already defined structures. Standardization Organizations employ democratic principles and processes in their technical committees (*TC*). Nevertheless considerable contributions to substantial parts of the *OASIS-DSS* specification have been made. We hope that additional points identified in this document may be considered by the OASIS Digital Signature Services eXtended (*OASIS-DSS-X*) *TC* for future versions of *OASIS-DSS*.

3.2 *OASIS-DSS* Basics

The chapter introduction presented *OASIS-DSS* as two request response protocols, one for signing and one for verifying. How *OASIS-DSS* could be used in a real life scenario is shown in Figure 3.1.1. On the left a signer sends an unsigned document, maybe some *XML* order form, to a *OASIS-DSS* server. Before the signed document is returned, the *OASIS-DSS* server for example decided on a certain signature format, necessary transforms and may have enclosed certain statements, policies and included the signing certificate. A copy of the data objects and the signature may have been stored on the server. The signed *XML* document is then sent off to the verifier.
The verifier receives this document by means of email or other electronic process. It may have traveled through several network nodes and potentially through various other higher level *XML* protocols.
The verifier then either verifies the document by herself or requests an *OASIS-DSS* server to perform verification for her, assuming on the receiver side is also a *OASIS-DSS* server present.
To the private key used for signing corresponds a public key, which is bound to the identity of the signer (signer's company). A certificate proving this, has been issued by a Certification Authority or trusted third party. Meaning a Public-Key Infrastructure (*PKI*) is available and allows to establish trust in the signer's public key.

OASIS-DSS is architecturally intended to be a stand alone service and can be layered on top of Hyper Text Transfer Protocol (*HTTP*) or other higher level protocols, such as SOAP (*SOAP*). They can be secured by Transport Layer Security (*TLS*) or Web Services Security (*WSS*)[1] respectively.
The *OASIS-DSS* specification supports a variety of signature formats:

- *XMLDSIG* [27],[29] aka. RFC 3275
- *CMS* (RFC 3852) Signatures (not covered in this thesis)
- RFC 3161 time-stamps (not covered in this thesis)
- *XML* time-stamps (defined in *OASIS-DSS*)
- In the Advanced Electronic Signatures Profile [61]
  - *XAdES ETSI* TS 101903 [46]
  - *CAdES ETSI* TS 101733 [47]

[1]Currently maintained by the Web Services Secure Exchange (*WS-SX*) *TC*.

The OASIS-DSS Request

Figure 3.2.1 shows a simple example of a <dss:SignRequest> requesting a detached *XMLDSIG* signature as will be further discussed in the following sections.

```
<dss:SignRequest xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema">
  <dss:OptionalInputs>
    <dss:SignatureType>urn:ietf:rfc:3275</dss:SignatureType>
  </dss:OptionalInputs>
  <dss:InputDocuments>
    <dss:Document RefURI="http://www.example.org/bar">
      <dss:InlineXML><ex:foo xmlns:ex="http://www.example.org/bar">
      <ex:bar/>
      <ex:baz/>
      </ex:foo></dss:InlineXML>
    </dss:Document>
  </dss:InputDocuments>
</dss:SignRequest>
```

Figure 3.2.1: An simple example of a <dss:SignRequest>

The example shows an optional input that designates the signature type and a very simple *XML* payload is provided as <dss:InlineXML>.

Namespaces and Types

The `targetNamespace`[2] used for elements and types is:

$$\texttt{urn:oasis:names:tc:dss:1.0:core:schema} \tag{3.2.2}$$

The *OASIS-DSS* specification [62] is built on a dss:RequestBaseType[3] and a dss:ResponseBaseType. The dss:RequestBaseType is extended by the local *anonymous types* in the element declarations of <dss:SignRequest> and the <dss:VerifyRequest>. Similarly the dss:ResponseBaseType is extended by the <dss:SignResponse>, the <dss:VerifyResponse> is inconsistent however and directly of type dss:RequestBaseType[4].
To handle those requests, a basic processing has been defined for signing and for verifying. It can be overridden and augmented by several optional inputs and outputs. To introduce *OASIS-DSS*, the optional parts will be ignored at first and many details skipped. The basic structures are introduced in the following sections, starting with the dss:RequestBaseType in subsection 3.2.1. The basic processing can be found in subsection 3.3.1. It defines the core functionality and can be extended or overridden by

[2]For `targetNamespace` recall subsection 2.3.4.

[3]Recall that elements will be shown as "<prefix:localname>", types as " prefix:localname " and attributes as `Attribute`, use also page viii.

[4]This may be an editorial problem; the <dss:VerifyResponse> has gone missing from the schema after cd-r04 (`http://lists.oasis-open.org/archives/dss/200611/msg00014.html`) and was differently reintroduced in wd49.

processing specified for variants of different payload or optional inputs. Some of them allow to enter the processing at later stages as explained in subsection 3.3.2.

Having explained how the payload is conveyed, its processing and signing can be found in section 3.3. Process entry and mutations for signing are in subsection 3.3.3. This approach explains the service and starts with what is important and successively goes into more detail. Unfortunately the specification is not in all parts written in such an order that makes this very order apparent.
Taking advantage of what has been established for signing, the differences when verifying are explained in section 3.4

3.2.1 dss:RequestBaseType

The dss:RequestBaseType in Figure 3.2.3 is a *complex type*. Recall from subsection 2.2.2 Figure 2.2.8 on page 12 that such can specify a content model consisting of structural description of the attributes and elements.

```
    <xs:complexType name="RequestBaseType">
155   <xs:sequence>
        <xs:element ref="dss:OptionalInputs" minOccurs="0"/>
157     <xs:element ref="dss:InputDocuments" minOccurs="0"/>
      </xs:sequence>
159   <xs:attribute name="RequestID" type="xs:string" use="optional"/>
      <xs:attribute name="Profile" type="xs:anyURI" use="optional"/>
161 </xs:complexType>
```

Figure 3.2.3: dss:RequestBaseType

The dss:RequestBaseType has two optional attributes. The `RequestID` attribute (line 159) is a general purpose string. The server echoes it in the response. The optional `Profile` attribute (line 160) indicates by means of a *URI* what profile should be used.
The structure of dss:RequestBaseType allows to omit <dss:OptionalInputs>. Surprisingly the same is true for <dss:InputDocuments>, the latter becomes clearer when we discuss the <dss:VerifyRequest> in Figure 3.4.1.

<dss:InputDocuments>

The <dss:InputDocuments> carries the payload. The <dss:OptionalInputs> are a means to supply instructions for amending and mutating the processing as will be discussed later. That the service is primarily tailored for the reference processing model of *XMLDSIG* can be seen from the payload container names <dss:Document>, <dss:TransformedData> and <dss:DocumentHash>.
<dss:Other> is a general purpose container for extensibility, allows *mixed content* from arbitrary namespaces and will be covered in section 3.7.
The <dss:InputDocuments> element specification in Figure 3.2.4 has a content model that is defined via an *anonymous type*. It is modeled as a sequence of unbounded choices. The sequence is redundant, but

```
<xs:element name="InputDocuments">
  <xs:complexType>
    <xs:sequence>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="dss:Document"/>
        <xs:element ref="dss:TransformedData"/>
        <xs:element ref="dss:DocumentHash"/>
        <xs:element name="Other" type="dss:AnyType"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 3.2.4: <dss:InputDocuments>

it may have been used as a stylistic element to clearly express that the unbounded occurrence of choices is in fact a sequence. The schema definition follows that a request will carry `1..n` input documents all of which will be covered by a <ds:Reference>.

<dss:TransformedData> enables a client to send a document sub-set and derived documents.

<dss:DocumentHash> carries only the fingerprint of a document to the server.

This allows service users and profile authors to split the burden of reference processing between the client and the server. A client can then enter the processing at later stages, what will be covered in more detail in subsection 3.3.2. For now we are interested only in the <dss:Document> and its dss:DocumentType which merely extends dss:DocumentBaseType and is hence not shown here.

```
<xs:element name="Document" type="dss:DocumentType"/>
```

Figure 3.2.5: <dss:Document>

3.2.2 dss:DocumentBaseType

The dss:DocumentBaseType is the base type for all forms of payload containers and supplies a set of attributes.

```
<xs:complexType name="DocumentBaseType" abstract="true">
  <xs:attribute name="ID" type="xs:ID" use="optional"/>
  <xs:attribute name="RefURI" type="xs:anyURI" use="optional"/>
  <xs:attribute name="RefType" type="xs:anyURI" use="optional"/>
  <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
</xs:complexType>
```

Figure 3.2.6: dss:DocumentBaseType

The `ID` attribute is provided, so that a payload container may be referred to by optional inputs and other protocol structures. The referring attributes in those structures are called `WhichDocument`.

The `RefURI` attribute (line 36) indicates what *URI reference* has been used to acquire the document. It will result in <ds:Reference>'s `URI` attribute (line 5 in Figure 2.4.1 on page 37). Depending on whether the `RefURI` attribute is either a *same-document reference* or an external *URI reference* (section 2.3.4 Figure 2.3.15) the contained data has to be either dereferenced to `NodeSetData` or `OctetStreamData` respectively. Despite its *Schema* declaration, the `RefURI` is not really an optional attribute, because it may be omitted on one child of <dss:InputDocuments> only.

The `RefType` attribute is equivalent to the `Type` attribute in <ds:Reference>, which intends to provide type information on the digest input, eventually resulting from resolving and transforming the data object.

The optional `SchemaRefs` are used to identify the *Schemas* to validate the data object on parsing.

3.2.3 dss:DocumentType - Forms of payload

The *OASIS-DSS* protocol provides a rich way of transporting the actual payload, which is usually comprised of documents or document equivalent representations. The forms of payload reflect the requirement for different fidelity modes when it comes to convey *XML* inside *XML*. *OASIS-DSS* supports different fidelity such as modes for binary, character or *XML* level equivalent transport.

```
<xs:complexType name="DocumentType">
  <xs:complexContent>
    <xs:extension base="dss:DocumentBaseType">
      <xs:choice>
        <xs:element name="InlineXML" type="dss:InlineXMLType"/>
        <xs:element name="Base64XML" type="xs:base64Binary"/>
        <xs:element name="EscapedXML" type="xs:string"/>
        <xs:element ref="dss:Base64Data"/>
        <xs:element ref="dss:AttachmentReference"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 3.2.7: <dss:DocumentType>

By default <dss:Base64XML> and <dss:Base64Data> should be used for *XML* and non-*XML* data objects respectively. Base64 Content-Transfer-Encoding (*base64 encoding*) allows for separation of payload from the transport protocol (cf. subsection 4.3.1).

<dss:EscapedXML> is an alternative to <dss:Base64XML>, that assures opaqueness on the character level. It may need less space if the ratio of *XML* tags to content is low.

<dss:InlineXML> uses *Exc-C14n* by default to have a well defined way for extracting a payload document from the transport protocol context. Nevertheless not all payload is suitable for *Exc-C14n* (cf. subsection 2.5.3.1).

Fidelity modes for conveying payload

Payload Container	Equivalence	Context
<dss:Base64XML>, <dss:Base64Data>	binary	none
<dss:EscapedXML>	character	encoding
<dss:InlineXML>	content	encoding, namespace, inheritable attributes

Figure 3.2.8: Payload in various forms

The most robust form for conveying payload in *XML* is *base64 encoding*, it preserves the binary equivalence for any kind of *XML* content. Hence the basic processing is specified for `Base64XML` first. It allows to transport, dereference and parse the conveyed data objects, without being affected by the surrounding encoding, namespace-context and other inheritable *XML* attributes, such as *xml:lang*, *xml:space* and *xml:base*. In turn expressions may also be in the document, which would evaluate differently in a context distorted by the surrounding transport protocol. The *base64 encoding* separates and shields the payload at the cost of additional serialization encoding and decoding plus parsing and of course the lost space efficiency and bandwidth consumption on the wire.

<dss:EscapedXML> uses the predefiened general character entities `&amp;`, `&lt;` and `gt;` to escape `&`, < and > where necessary[5]so it matches the `CharData` production [6] of *XML*. <dss:EscapedXML> is already forced to use at least the same character encoding as the surrounding transport protocol, but is meant to assure characterwise equivalence, potentially trans-coded from its original character encoding. This already requires the corresponding <ds:References> to be made robust against a change in the encoding, especially if referenced by an external *URI* (Figure 2.3.15) and if no <ds:Transforms>, causing either an implicit (subsection 2.5.2) or an explicit canonicalization of the data are applied. Recalling the overview given in Figure 2.5.3, the bottom line for canonicalizing data objects that travel in <dss:EscapedXML> over systems that use different encodings is *MC14n* (subsection 2.5.1). Nevertheless <dss:EscapedXML> is essentially a string that contains an escaped *XML* document. No context is inherited with respect to *XML* technologies. The costs remain the same as with `Base64XML`.

In contrast, when using <dss:InlineXML> it can be handled in the same parsing step, but one has to be considerate of the entire surrounding context. Like with <dss:EscapedXML>, it is not possible to send data in <dss:InlineXML> that has another character encoding than the surrounding *OASIS-DSS* transport protocol. The content shall further not depend or be affected by the existence of namespace-declarations and other inheritable attributes in scope, otherwise the signatures may break. Hence *OASIS-DSS* mandates to use *Exc-C14n* to extract the data object. As seen in subsection 2.5.3.1 not all data is suitable for *Exc-C14n* (see also subsection 4.3.1). Applications not using *xml:base* or *simple inheritable*

[5]If no `CDATA` sections are used it is sufficient `]]>` also has to be escaped to `]]&gt;`. *OASIS-DSS* is not explicit about this.

attribute like *xml:lang* and *xml:space* or working with connected node-sets and having no *QNames* in content will not experience any problems. Subsection 4.3.1 will discuss this in more detail.

`Base64Data` is intended to be used for non *XML* data, usually external to the <ds:Signature>, which implies that the `RefURI` and respectively the <ds:Reference>'s `URI` have to be external *URI references*. If the `RefURI` is a *same-document reference*, the data object has to be included into the same document as the resulting signature. Otherwise it cannot be dereferenced. The latter is solved in *OASIS-DSS* by mandating to use the optional input <dss:IncludeObject> in such a case (cf. Figure 3.3.5).

3.3 Signing

Having established in the previous sections how payload arrives in a <dss:SignRequest> at an *OASIS-DSS* server and what fidelity modes exist, the following sections describe how they are processed for signing. <dss:Base64XML> is what a client should use by default, as it provides the highest fidelity for payload transport. With <dss:EscapedXML>, used with external *URI references*, care has to be taken by a client or the server that it is at least canonicalized with *MC14n*. <dss:InlineXML> has also been introduced, details of its processing are however deferred to keep the explanation of the basic processing explanation simple.

3.3.1 Basic processing for signing using *XMLDSIG*

The basic processing deals with the various forms of payload and reflects *XMLDSIG's* core and reference processing model. As mentioned previously, the processing may be entered at various stages, ranging from dereferencing a document, over the digest input up to the digest value. This makes the basic processing very generic and it can be adapted, extended or overridden by optional inputs and profiles in a flexible way.
The most common use case is when a client supplies payload in form of a plain <dss:Document>. The normative assumption in this case is that one <dss:Document> within <dss:InputDocuments> corresponds to one <ds:Reference>.
As described in subsection 2.3.2 an *XML* document has a *document order*. This applies in principle also to the <dss:InputDocuments>, but a server *MAY* ignore this order. Thus a server can respect the order but may not assume other implementations respect it as well. This mainly plays a role for processing a <dss:VerifyRequest>. It implies that a server will have to be prepared that the <ds:Reference> elements in a <ds:Signature> may be in a different order than in the <dss:InputDocuments>.

3.3.2 Enter the processing at various stages

Having only dealt with <dss:Document> so far, we are now coming back to Figure 3.2.4 of section 3.2.1 on page 56. The various stages at which the basic processing can be entered will be discussed. The payload is supplied in <dss:InputDocuments> as either <dss:Document>, <dss:TransformedData>

```
1. FOR EACH <dss:Document> IN <dss:InputDocuments>
   a. IF <dss:Base64XML> (see later sub-sections for other cases)
      THEN base64-decode, date MUST be a well formed XML Document
      OTHERWISE ERROR
      IF RefURI IS same document reference THEN parse to NodeSetData.
   b. Server MAY CHOOSE to apply additional <ds:Transform>s
   c. CHOOSE a <ds:DigestMethod> and calculate the <ds:DigestValue>
   d. CREATE a <ds:Reference> as follows:
      i.   IF a RefURI attribute exists THEN
             <ds:Reference>'s URI = RefURI
           ELSE URI is omitted.
           (More than one RefURI omitted is an error).
      ii.  IF a RefType exists THEN ds:Reference's Type = RefType
      iii. SET ds:Reference's <ds:DigestMethod> as used in c.
      iv.  SET ds:Reference's <ds:DigestValue> as calculated in c.
      v.   SET ds:Reference's <ds:Transforms>, the sequence MUST
           describe the effective transform as a reproducible
           procedure from parsing until hash.

2. ds:Reference elements resulting from optional inputs
   MUST be included.
3. CREATE an XML signature using the ds:Reference elements created
   in Step 1.d, according to [XMLDSIG].
```

Figure 3.3.1: Simplified Basic Processing

or <dss:DocumentHash>. These match the following terms `DereferencedData`, `DigestInput` of *JSR105*[6] and <ds:DigestValue> of *XMLDSIG*, which is summarized in Figure 3.3.2.

XMLDSIG, *JSR105*	*OASIS-DSS*
<ds:Reference> `URI`	<dss:DocumentBaseType> `RefURI`
⇓	⇓
`DereferencedData`	<dss:Document>
⇓	⇓
`DigestInput`	<dss:TransformedData>
⇓	⇓
<ds:DigestValue>	<dss:DocumentHash>

Figure 3.3.2: The reference processing can be split between clients and severs

<dss:Document> matches the locations addressable by a *URI reference* (3. in Figure 2.4.5 on page 41). Using a *JSR105* compliant implementation such as the XML Security Toolkit (*XSECT*) a custom server-side `URIDereferencer` can be employed to dereference the data objects from within the <dss:InputDocuments>.

[6]Java Specification Request 105 XML Digital Signature APIs (*JSR105*)

<dss:TransformedData> inside <dss:InputDocuments> splits reference processing between the client and server. The basic processing is entered before a <ds:DigestMethod> is chosen in 1.c. in line 8 of Figure 3.3.1. This corresponds to one of the edges labeled with `OctetStreamData` entering the DigestMethod (1.1.4 Figure 2.4.5 on page 41) in the *XMLDSIG* reference processing model. In short the `DigestInput` is supplied in terms of *JSR105*. The *JSR105 API* unfortunately did not allow to supply a `DigestInput` on reference creation. Hence the extension of the *API* was requested during the work on this thesis.
<dss:DocumentHash> provides a means to perform all of *XMLDSIG's* reference processing, excluding the actual formation of the <ds:Reference> on the client side. *JSR105* again did not allow to supply a <ds:DigestValue> on reference creation and needed to be extended as well.
The element declaration of <dss:Document> was in Figure 3.2.5 on page 56, now the element declarations for <dss:TransformedData> and <dss:DocumentHash> follow.

3.3.2.1 <dss:TransformedData>

This form of payload extends <dss:DocumentBaseType> and informs the *OASIS-DSS* server what <ds:Transforms> have been applied by the client already.

```
<xs:element name="TransformedData">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:DocumentBaseType">
        <xs:sequence>
          <xs:element ref="ds:Transforms" minOccurs="0"/>
          <xs:element ref="dss:Base64Data"/>
        </xs:sequence>
        <xs:attribute name="WhichReference" type="xs:integer"
           use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Figure 3.3.3: <dss:TransformedData>

<dss:Base64Data> underlines the intention that the conveyed data is to be digested directly and must be in *base64 encoding*.
The `WhichReference` attribute is added and will be of significance in a <dss:VerifyRequest> and hence ignored at first.

Why <dss:TransformedData>?

The question could be asked: If the client performed all the transforms why shouldn't it also perform the digest computation?

The main reason is that the burden of choosing a <ds:DigestMethod> is taken off the client. A corporate sever may have to enforce certain policies or even choose different cryptographic hash functions depending on the context of the data or addressee to fulfill certain business logic, yet giving the client the chance to use *XML* technologies to derive the data from potentially complex sources.
It further allows servers to inspect and archive what has been signed, what might be required in the case when corporate keys are used. Whether a server would want to vouch for some digest value without having the chance to inspect the data is a legal or organizational issue and out of the scope of this document and may be a topic for future work.
A more advanced use case is, when a client performs requests to some database and uses *XML* technologies to format the results and eventually produces an *XHTML* document. This may have to fulfill a certain format, e.g. some corporate tax declaration. The client wants to get such a document signed and maybe timestamped by some authority running an *OASIS-DSS* server. So the client can prove to have produced the declaration and also shows how it was produced. In the case of an audit the authority can then later check if the production of the data in the environment of the client is reproducible.
So the dereferencing from data sources is performed at the client-side, as is the compilation of this data up to the final report. The production and formatting of this report is documented in the chain of transforms by means of the <ds:Transfoms> and only the derived data is then sent off to the server, allowing a client to avoid storing the actual document, yet prepared for a later verification and reproducibility of the document.

Processing <dss:TransformedData>?

For simplicity in the basic processing it was decided that <dss:TransformedData> would be directly hashed by omitting step 1.b. (section 3.3.5 [62]). All the <ds:Transforms> have been applied by the client already.
<dss:TransformedData> enters reference processing at the end of the chain of <ds:Transforms> and has to convey `OctetstreamData` (200602/msg00035)[7]. This implies that some form of canonicalization or transformation that results in `OctetStreamData` has to be the last element child of the <ds:Transforms> element (line 74 in Figure 3.3.3).

Why should the server not apply additional transforms in the basic processing?

Transmitting intermediate results from a client to a server is unspecified, especially in the case of `NodeSetData` that does not comprise a complete document.
Only `OctetStreamData` can be supplied and `NodesetData` requires the complete document to be transmitted in the general case. In the cases where `NodeSetData` is in terms of the *Infoset* self con-

[7]This is a shorthand notation to point to discussion list entries on which the modifications to *OASIS-DSS* were based. They are links and can be clicked in the electronic version of this document. The *URI* can also be gathered by prepending 'http://lists.oasis-open.org/archives/dss/' and appending '.html'.

tained, there is no reason to apply further transforms or canonicalization by the server. For transmission the `NodesetData` needs to be serialized and the serialization method is precisely *C14n*.
So the server can safely assume the `NodeSetData` was implicitly canonicalized by *C14n*.
If the client would not do this and terminate the <ds:Transforms> with a <ds:Transform> that results in `NodeSetData` and not use *C14n* for implicit serialization, the result is undefined.
An *OASIS-DSS* server would not accept such an implicit serialization-parsing step due to transmission in the middle of the chain of transforms, because then the requirement of the basic processing: *“. . . , the sequence MUST describe the effective transform as a reproducible procedure from parsing until hash . . . ”* [62] would be violated.
Hence the basic processing for <dss:TransformedData> assumes that data is to be digested directly. It may be overridden however by optional inputs (e.g. <dss:SignedReferences> subsection 3.3.6), or by profiles or their optional inputs.
And last but not least it will allow a smart server implementation to directly consume the supplied data on first parsing by applying the hash function and to save so memory.

3.3.2.2 <dss:DocumentHash>

<dss:DocumentHash> assumes the reference processing including the digest calculation has been performed at the client side. It informs an *OASIS-DSS* server what <ds:Transforms> have been applied, what <ds:DigestMethod> was used and the <ds:DigestValue> is conveyed as well. All of *XMLDSIG’s*

```
<xs:element name="DocumentHash">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:DocumentBaseType">
        <xs:sequence>
          <xs:element ref="ds:Transforms" minOccurs="0"/>
          <xs:element ref="ds:DigestMethod" minOccurs="0"/>
          <xs:element ref="ds:DigestValue"/>
        </xs:sequence>
        <xs:attribute name="WhichReference" type="xs:integer"
           use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Figure 3.3.4: <dss:DocumentHash>

reference processing, excluding the actual formation of the <ds:Reference> is here performed on the client side. This may be useful if in some use case, where the digest value of some data object is well known and often included so the server can accept it as a “known good” value.
In many application contexts - like when a corporate key is used - simply vouching for some hash value is not a good idea and “known good” values are essential, because: *Only What is ”Seen” Should be Signed* [8][27][29].

This principle formulates in one sentence that obviously what has been discarded in <ds:Transforms> is not signed and that a <ds:Transform> cannot be trusted in general. It however also applies in the reverse direction of a hash function, signing a hash value might cause you to be committed to what this seemingly arbitrary sequence of bits might represent as a fingerprint.
It is a principle that one should not sign what is not understood. This is also true for a hash value which by its preimage resistance cannot be understood by definition, unless it is known to belong to a certain input or document.
In another potential use case where the authenticated signer wishes to use his own key, that is maybe stored encrypted on an *OASIS-DSS* server, the reverse may be true. In such a situation sending only a hash would help to ensure the signers privacy.
If such a system can be trusted or whether a server would want to vouch for some digest value without having the chance to inspect the data is an organizational or a legal issue and out of the scope of this thesis.
On verification the situation is far less critical and may be an enabler for privacy protection and performance increases in successive requests, always containing signatures with some often appearing <ds:Reference>.

To summarize, *OASIS-DSS* has different fidelity modes for carrying payload allows to enter *XMLDSIG's* reference processing at various stages.

3.3.3 Important optional inputs and outputs for signature creation

OASIS-DSS also supports a set of optional inputs and related outputs to handle a variety of signature topographies like enveloping, enveloped and detached *XMLDSIG* signatures. They are explained in subsection 2.4.1.2.
The optional inputs as <dss:IncludeObject> and <dss:SignaturePlacement> can be understood as commands to a signature creation engine to alter the basic processing described earlier in subsection 3.3.1 to create enveloping or enveloped <ds:References>. One will note that the terms are used here with respect to a <ds:Reference> and not with respect to the <ds:Signature>. This is more appropriate, because a signature can be detached, have a signed <ds:Object>, and be placed inside an *XML* document that it signs at the same time.

3.3.4 <dss:IncludeObject> - Creating enveloping signatures

<dss:IncludeObject> is used to create enveloping signatures and refers via the the `WhichDocument` attribute of type `xs:IDREF` to a <dss:Document>. This document will be placed in a <ds:Object> of the resulting <ds:Signature>. This optional input makes no sense for <dss:TransformedDocument> and <dss:DocumentHash> as placing signatures into transient and transformed documents is not useful. The server either creates the enveloping <ds:Object> or grants the client to send a prepared one by setting `hasObjectTagsAndAttributesSet` (200503/msg00007). This takes the burden of the

```
<xs:element name="IncludeObject">
  <xs:complexType>
    <xs:attribute name="WhichDocument" type="xs:IDREF"/>
    <xs:attribute name="hasObjectTagsAndAttributesSet" type="xs:boolean"
        default="false"/>
    <xs:attribute name="ObjId" type="xs:string" use="optional"/>
    <xs:attribute name="createReference" type="xs:boolean"
        use="optional" default="true"/>
  </xs:complexType>
</xs:element>
```

Figure 3.3.5: <dss:IncludeObject>

server to invent `ID` attributes.

The attribute `createReference` triggers the creation of a <ds:Reference> covering the document in question, and is by default set to true.

An enveloping signature bears the data object covered by at least one of their references. This implies that the *URI* attribute of the corresponding <ds:Reference> has to be a *same-document reference*.

In *OASIS-DSS* WD30 page 22 the optional input was called <dss:EnvelopingSignature> and was seriously underspecified.

3.3.5 <dss:SignaturePlacement> - Creating enveloped signatures

<dss:SignaturePlacement> is used to place the resulting <ds:Signature> inside a document which is conveyed in <dss:Document> and causes the creation of a <ds:Reference> with *same-document reference* by default.

```
<xs:element name="SignaturePlacement">
  <xs:complexType>
    <xs:choice>
      <xs:element name="XPathAfter" type="xs:string"/>
      <xs:element name="XPathFirstChildOf" type="xs:string"/>
    </xs:choice>
    <xs:attribute name="WhichDocument" type="xs:IDREF"/>
    <xs:attribute name="CreateEnvelopedSignature" type="xs:boolean"
        default="true"/>
  </xs:complexType>
</xs:element>
```

Figure 3.3.6: <dss:SignaturePlacement>

This <dss:Document>'s `RefURI` is required to include the document by a *same-document reference* and normally the use of the "Enveloped Signature Transform" (section 6.6.4 [27] [29]) is indicated as well. The "Enveloped Signature Transform" makes use of the `here()` function [27] [29] and assures that the signature does not sign itself. The `here()` function and thus the "Enveloped Signature Transform" is only effective if it is not preceded by some transform emitting `OctetStreamData`. It is

necessary to be evaluated in the "same" document.
Finally the enveloped signature is returned in the optional output <dss:DocumentWithSignature> (a container for a <dss:Document>, Figure 3.3.7). *OASIS-DSS* does not specify what fidelity mode should be used for returning the signature. It seems to be a fair assumption to choose the same mode as has been used for sending the input document to bear the signature to the server.

```
<xs:element name="DocumentWithSignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dss:Document"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 3.3.7: <dss:DocumentWithSignature>

3.3.5.1 Client side splicing

The concept of client-side splicing used to be in an earlier working draft (WD) of *OASIS-DSS* (i.e. WD30 page 16 and WD30 page 20). There the server would have to compute the signature over data objects that are out of their context. Even all sorts of *same-document references* and <ds:Transforms> would be evaluated in multiple node-sets, each comprising a sub-document, that would have to be independent of the remaining document that has not been sent by the client. The evaluation would have to be performed as if it were one document.

Client-side splicing of signatures, although an interesting concept to save bandwidth on the back-channel from the server, was deferred to a profile by the *OASIS-DSS TC*. It would have required normative language to be added to the core document, which however would have to be fulfilled by clients. *OASIS-DSS* however only defines processing for the server and normative text defining the behaviour of the client would not really fit into the core specification. Further an *XMLDSIG* profile would have been needed and blown up the document. All this seems to be best addressed in a document next to the core document, but not in it. Unfortunately a concrete proposal for such a profile has since then never been made or pursued by anyone (200507/msg00042, 200508/msg00007).
It would potentially be an interesting topic for future standardization.

3.3.6 <dss:SignedReference> - More control on reference generation

The <dss:SignedReferences> contains one or more <dss:SignedReference> elements and may be supplied as optional input in a <dss:SignRequest>. The one-to-one mapping of the basic processing from input documents to created <ds:Reference> elements can be overridden. This is done by supplying multiple <dss:SignedReference> elements pointing to the same input document with the `WhichDocument` attribute.

If there is a `RefURI` present in the optional input, then an additional <ds:Reference> is created. The `RefURI`'s value is set assigned to the <ds:Reference>'s `URI`, the `RefId` sets the <ds:Reference>'s `ID` and the <ds:Transforms> are executed by the server. The server may perform zero or more additional <ds:Transforms>, and if needed canonicalize the result. Eventually the <ds:Digestvalue> will be calculated and the transforms set in the <ds:Reference>.

If the `RefURI` is not set - then normal reference processing for the pointed to <dss:Document> is adapted accordingly by overriding `RefId` and <ds:Transforms>.

<dss:SignedReferences>'s <ds:Transforms> do not make any sense with <dss:DocumentHash> and *OASIS-DSS* mentions this optional input only with <dss:Document>.

```
<xs:element name="SignedReference">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ds:Transforms" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="WhichDocument" type="xs:IDREF" use="required"/>
    <xs:attribute name="RefURI" type="xs:anyURI" use="optional"/>
    <xs:attribute name="RefId" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
```

Figure 3.3.8: <dss:SignedReference>

The *OASIS-DSS* core does not define what happens when <dss:SignedReference> is used in combination with a <dss:TransformedData> in a <dss:SignRequest>. *[...] 1. The server identifies the `<Document>` referenced as indicated by the `WhichDocument` attribute. [...]* [62] A potential interpretation could be however, that the server refers to *the `<Document>`* in a more general sense and *OASIS-DSS* rather means an element of <dss:DocumentBaseType>.

Then would also follow from step 2. 1.b. of <dss:SignedReferences>'s processing: *[...] the server may apply any other transform it considers appropriate as per its policy [...]* [62]. The input would be `OctetStreamData` and require the server to add a *C14n* before the first <ds:Transform> executed on the server-side, if the last of <dss:TransformedData>'s <ds:Transforms> executed by the client ended in node-set data. The latter is implied by an implicit *C14n* at the client side to serialize and transmit the data, which would have to be reflected as per *OASIS-DSS'* requirement for effective reflection of the true transform[8].

To specify such subtleties would have blown up the core document and further explains why additional server-side transforms are not allowed for <dss:TransformedData> in the basic processing section 3.3.2.1 on page 62.

Given that the processing may be entered in the middle of the <ds:Transforms> using *OASIS-DSS*, it is important that it can be implemented using a standard *API* such as *JSR105* as well. The extension to

[8]As *C14n* is idempotent it would be safe to canonicalize anyway and not to rely on the clients choice for serialization.

JSR105 API, mentioned earlier would however be able to cope with the scenario of intermediate result transmission.

3.3.7 Further optional inputs and outputs for signature creation

The optional input <dss:IntendedAudience> enables a specific instance of an *OASIS-DSS* server to make choices and decisions based on the identity of the recipient.
The <dss:KeySelector> is a wrapper for a <ds:KeyInfo> allowing a client to tell an *OASIS-DSS* server instance which key should be used for signing.
The < dss:Properties > are grouped in two lists.

- < dss:SignedProperties >
- <dss:UnsignedSignedProperties>

They contain <dss:Property> elements having an identifier and an associated value of arbitrary *mixed content*. The Advanced Electronic Signature Profiles [61] make extensive use of this optional input to include signed properties.

3.3.8 Other optional inputs

Other optional inputs that may be used during signing, but also for verifying, like for accessing the service, policy matters or providing auxiliaries are mentioned in the following paragraph.
The optional input < dss:ServicePolicy > helps to establish a common expectation on a service policy. < dss:ClaimedIdentity > can be used to supply a name; the claim can be supported by checking it against a secure underlying transport binding (e.g. *TLS* with client certificate, *TLS* server authentication with *HTTP* password protection) or message level authentication like an assertion as specified in the Security Assertion Markup Language (SAML). *OASIS-DSS* is not specific on how this identity is used.
The optional input <dss:Schemas> allows to convey a list of *Schemas* and takes advantage of the dss:DocumentType to be used for parsing the <dss:InputDocuments>. <dss:Schemas> may be used as optional outputs as well as the *Schema* of a signed document can differ from the *Schema* of an unsigned document, especially with enveloped signatures. Applications sometimes discover the need for securing their documents and messages after systems have been deployed in which case *Schema* cannot be changed and the addition of a <ds:Signature> could cause validation errors.

3.3.9 dss:ResponseBaseType - Returning signed documents

The dss:ResponseBaseType (Figure 3.3.9) defines how a response reports on the success of the processing by using a <dss:Result> and provides a container called <dss:OptionalOutputs>.
The `OptionalOutputs` are of significance for returning enveloped signatures and have already been mentioned in the context of <dss:SignaturePlacement> and optional <dss:Schemas> output.

```
<xs:complexType name="ResponseBaseType">
  <xs:sequence>
    <xs:element ref="dss:Result"/>
    <xs:element ref="dss:OptionalOutputs" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="RequestID" type="xs:string" use="optional"/>
  <xs:attribute name="Profile" type="xs:anyURI" use="required"/>
</xs:complexType>
```

Figure 3.3.9: dss:ResponseBaseType

The <dss:Result> contains a <dss:ResultMajor> and a <dss:ResultMinor> element of simple type `xs:anyURI`[9]. <dss:ResultMajor> values are:

- `urn:oasis:names:tc:dss:1.0:resultmajor:` followed by:
  - `Success`
  - `RequesterError` - indicating some error on the part of the client.
  - `ResponderError` - indicating some error on the part of the server.
  - `InsufficientInformation` - request failed due to a lack of information form some other party or service.

The <dss:ResultMinor> qualifies the result further and provides the outcome or the actual reply of the service. Its value always has the prefix `urn:oasis:names:tc:dss:1.0:resultminor:` and is in the case of a <dss:SignResponse> followed for example by:

- `urn:oasis:names:tc:dss:1.0:resultminor:` followed by:
  - `MoreThanOneRefUriOmitted` - indicating the corresponding violation of the *XMLD-SIG* reference processing model (cf. subsection 2.4.1.3).
  - `InvalidRefURI`
  - `NotParseableXMLDocument`
  - `NotSupported` - for unsupported optional inputs.
  - ...

3.3.10 <dss:SignResponse>

A <dss:SignResponse> (Figure 3.3.10) extends the dss:ResponseBaseType and additionally returns a <dss:SignatureObject> (Figure 3.3.11 on page 70) containing either an in-line <ds:Signature> or

[9]We recall that *Schema* [20] functions as a type donor, in such a case the prefix "`xs:TypeName`" is commonly used to refer to named top level types defined directly in *Schema*.

a `Base64Signature` in *base64 encoding*. This only works with enveloping or detached signatures. Alternatively a pointer to an enveloped signature called < dss:SignaturePtr > referring into a <dss:DocumentWithSignature> is provided.

```
<xs:element name="SignResponse">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:ResponseBaseType">
        <xs:sequence>
          <xs:element ref="dss:SignatureObject" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Figure 3.3.10: <dss:SignResponse>

```
<xs:element name="SignatureObject">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element ref="ds:Signature"/>
        <xs:element ref="dss:Timestamp"/>
        <xs:element ref="dss:Base64Signature"/>
        <xs:element ref="dss:SignaturePtr"/>
        <xs:element name="Other" type="dss:AnyType"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
  </xs:complexType>
</xs:element>
```

Figure 3.3.11: <dss:SignatureObject>

<dss:Other> is a general purpose container to allow for extensibility.

Section Summary

To summarize *OASIS-DSS* receives a <dss:SignRequest> and returns a <dss:SignResponse>. The optional inputs <dss:IncludeObject> and <dss:SignaturePlacement> can be used for the creation of enveloping and enveloped signatures respectively.

<dss:SignaturePlacement> causes the enveloped signature to be returned in an optional output of the <dss:SignResponse> called <dss:DocumentWithSignature>. The newly created <ds:Signature> will be pointed to by the returned <dss:SignatureObject>'s child element < dss:SignaturePtr >.

3.4 Verifying

The <dss:VerifyRequest> extends dss:RequestBaseType (subsection 3.2.1) and inherits the attributes `Profile`, `RequestID` and the child elements <dss:OptionalInputs> and <dss:InputDocuments>. Hence it can just as well convey payload in various forms from *base64 encoding* in <dss:Base64XML> to <dss:InlineXML> and enter the processing at various stages.

The <dss:SignatureObject> (Figure 3.3.11 on page 70) appears in the <dss:SignResponse> and in the <dss:VerifyRequest> (Figure 3.4.1). Like in the <dss:SignResponse> it contains either the signature itself or a pointer into one <dss:Document>.

Recall from dss:RequestBaseType that <dss:InputDocuments> were optional, which becomes clearer now as they may be "substituted" by <dss:SignatureObject> for verifying enveloping <ds:Signature>s. However this does not only make the service less clear, we would have preferred an approach where enveloping signatures are transported in a <dss:Document> because of context free extraction and symmetry.

```
<xs:element name="VerifyRequest">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:RequestBaseType">
        <xs:sequence>
          <xs:element ref="dss:SignatureObject" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Figure 3.4.1: <dss:VerifyRequest>

For streaming processing of <ds:Reference> elements <ds:Transforms> and <ds:DigestMethod> must be accessible prior to the actual payload. Hence it would be preferable if <dss:SignatureObject> containing this information appeared before the <dss:InputDocuments>. However *Schema* can extend content types only at their content model's end [20].

<dss:SignatureObject>'s <dss:SignaturePtr> enables a requester to place the signature or signed document into a <dss:Document> child of <dss:InputDocuments>. <dss:SignaturePtr> is not useful for <dss:TransformedData> and <dss:DocumentHash>.

The <dss:SignatureObject> including <dss:SignaturePtr> may even be omitted, in which case the basic processing assumes only one <dss:Document> that must contain the signature[10]. This is perhaps a too harsh constraint as it prevents detached signatures from being verified with this method. Allowing only one <dss:Document> hinders necessary data objects from being sent. A weaker constraint as for example to only verify the first signature from the first input document would be sufficient. This

[10]Interestingly this case has been reserved for multiple signature verification optionally supported in the core.

however is not compatible with the view that the order of <dss:InputDocuments>' children should not matter (Figure 3.3.1).
A streaming implementation could nevertheless proactively calculate each <ds:Reference> it discovers, assuming it points to nodes that follow in document order.

3.4.1 Basic processing for verifying using *XMLDSIG*

The allegedly simplest case is a <dss:SignatureObject> containing a detached or enveloping signature that shall be verified. Otherwise `SignaturePtr` will refer to a <ds:Signature> enveloped in some document. As noted, <dss:SignatureObject> is optional, and if omitted this represents a variant of the basic processing for verifying *XMLDSIG* signatures. In this case only a single input document is permitted and the *OASIS-DSS* server must find all <ds:Signature>s contained in the document. Then either verify them all or respond with an error if multiple signatures are not supported.

As signatures conveyed inside <dss:SignatureObject>'s <ds:Signature> constitute some sort of inline *XML* payload, the same considerations as with <dss:InlineXML> apply (see also section 3.2.3). So although it intuitively seems to be a simple case it is advisable to prefer `SignaturePtr`. The latter will allow to take advantage of the different forms of payload.
Otherwise *OASIS-DSS'* reference processing for verifying *XML* signatures is very similar to the basic processing for signature creation.
It is also possible to enter it at various stages (cf. subsection 3.3.2) which works just like the processing of <dss:SignRequest>. The basic processing for verifying *XMLDSIG* has one more finesse to offer with respect to processing stages: The case of <ds:Reference>' `URI` and `Type` matching multiple elements (<dss:Document>, <dss:TransformedData> or <dss:DocumentHash>) in Step 2. d. If there is a <dss:Document> in this set an error is issued. If the set only consists of <dss:TransformedData> and <dss:DocumentHash> elements then an additional flag `WhichReference` must disambiguate.
This should avoid the case where the responsibility for processing the chain of transforms for one document is on the server-side and at the client-side at the same time. A client could falsely expect that the server would also match all the chain of transforms against those in the <ds:References> to find the matching one, which would be too expensive. Or that the sever would check all those for consistency as well. Usually there is a one to one mapping from *URI* to <ds:Document> to <ds:Reference> however.

3.4.2 Important optional inputs and outputs for signature verification

<dss:VerifyManifests> cause the verification of <ds:Manifest>. A <ds:Manifest> is a collection of <ds:Reference>s like a <ds:SignedInfo>. The manifest is usually referred to by a <ds:Reference> of the <ds:SignedInfo>. Manifest verification in *XMLDSIG* differs from <ds:SignedInfo> verification in two major points:

- It is not part of *XMLDSIG* core verification.

- some of the <ds:Reference>s in the <ds:Manifest> may fail, depending on the application.

This requires verbose reporting as specified in the < dss:VerifyManifestResults >, which reports on the verification of each <ds:Reference>.
The <dss:UseVerificationTime> optional input is of significance for *PKI* path validation. I.e. checking of the validity periods of all certificates up to the trust anchor.
The < dss:ReturnVerificationTimeInfo > causes the < dss:VerificationTimeInfo > to be returned, which states what time was actually used for verifying the signature.

3.4.3 <**dss:VerifyResponse**>

A <dss:VerifyRequest> is answered by a <dss:VerifyResponse>.

```
<xs:element name="VerifyResponse" type="dss:ResponseBaseType"/>
```

Figure 3.4.2: <dss:VerifyResponse>

The <dss:VerifyResponse>'s `ResultMajor` just works like the one of the <dss:SignResponse> and is inherited from dss:ResponseBaseType in subsection 3.3.9 on page 68.
A <dss:VerifyResponse> just reports on the outcome of verification and does this by means of a `ResultMinor` *URI*.
The prefix `urn:oasis:names:tc:dss:1.0:resultminor:` is followed by either:

- `valid:signature:OnAllDocuments`
- `valid:signature:NotAllDocumentsReferenced` - indicating that some of the input documents were not referenced.
- `invalid:IncorrectSignature`

3.5 Signing, Verifying - concluding remarks

A <dss:SignResponse> is in terms of its payload similar to a <dss:VerifyRequest>, but different with respect to their base type. The question arises why the protocol is not symmetric with respect to document payload. Having a <dss:PayloadDocuments> as the pendant to <dss:InputDocuments> would allow <dss:SignRequest>, <dss:SignResponse> to share one type make the *Schema* smaller and easier to implement. This triple could extend a type potentially called dss:MessageWithPayloadType and extend a dss:MessageType. The <dss:VerifyResponse> could extend the latter and <dss:Result> could be added to <dss:SignResponse> and <dss:VerifyResponse> as a facet.
Nevertheless profiles can take advantage of the current asymmetry with client-side splicing (subsection 3.3.5.1). A client side-splicing profile would have to address concerns with respect to only signing

what is seen [8][27][29] and constraining technologies such as *XPath*, *XPointer* and *XSLT* so that they will behave correctly if evaluated on sub-documents instead of the complete document.

The *OASIS-DSS TC* hence decided not to build the basic processing on client-side splicing to avoid false negatives and spurious validation errors. An *OASIS-DSS* server may even verify a signature outside the *OASIS-DSS* context before returning it, able to warn a client if bad input had been specified. This avoids that clients produce arbitrary signatures not verifiable outside the *OASIS-DSS* protocol. Nevertheless profiles have the freedom to use client side-splicing.

3.6 *OASIS-DSS* Profiles

OASIS-DSS provides a way to profile and extend the core protocol and there is a requirement for extensibility. *OASIS-DSS*-profiles are not profiles in the classical sense as they are not only constraining the *OASIS-DSS* core but also extending it by adding optional inputs and outputs.

There are profiles for *OASIS-DSS*, but implementation support for them is hard to be estimated at this point in time.

The profiles have not been examined in detail for correctness or compliance. This section just gives an overview on how profiles constrain or extend the core.

A profile like the "German Signature Law Profile" is written in a very simple way and puts the requirement of legal conformance with the German signature law on a server supporting this profile. If and how this requirement can be fulfilled in a client server architecture is not specified there and merely a reference to the law is provided.

Otherwise this profile requires, that only <dss:Document> payload is to be used. An identifier for the `ProfileID` has been defined and an optional input called <gsl:SignerRole>[11] to transport attribute certificates. Optional outputs have been defined and the <dss:InputDocuments> must be returned (echoed) as optional output.

Other and more complex profiles are the "Advanced Electronic Signature Profiles", a document for creating *XAdES* and *CAdES* signatures on an *OASIS-DSS* server. In these profiles the < dss:Properties >'s < dss:SignedProperties > have been used extensively to communicate what *XAdES* or *CAdES* properties should be added to the signature that is to be created. A new optional input <ades:SignatureForm> has been defined to request signatures ranging from a basic electronic signatures, over such including policies, timestamps, properties for long term verification and archival formats [46]. The latter could be seen as an electronic signature packaged with verification information, which was cached during a verification. Finally the signature in archived form is timestamped. For more details refer to [46][48].

[11] Assuming "`gsl:`" would be the prefix associated with the profiles namespace.

3.7 *OASIS-DSS* protocol extension points

The eXtensibility in *XML* (recall subsection 2.1.1) stands for the general principle to ignore [10] markup not understood by an application logic and to continue to process the interpretable data[12].
The *WSS TC* [63] explicitly did not want such “new” markup to be just ignored. A security language has unique requirements and the consequences of ignored data can be severe [10]. *OASIS-DSS* in contrast considers this to be implied and is not as explicit as *WSS*. *OASIS-DSS* provides the <dss:ResultMinor> value `urn:oasis:names:tc:dss:1.0:resultminor:NotSupported` for markup that has not been recognized. Whether profiles may specify uncritical optional inputs or what happens by default is not specified.
OASIS-DSS uses *Schema* and a validating *OASIS-DSS* processor will only allow extensions in designated locations specified in the *Schema*. These designated locations are called wild-cards.

```
<xs:complexType name="AnyType">
  <xs:sequence>
    <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Figure 3.7.1: dss:AnyType

The wild-card `xs:any` (Part 1 section 3.10.2 [20]) is redefined as a type called dss:AnyType in *OASIS-DSS* shown in Figure 3.7.1. This type is used to match optional inputs and optional outputs (Figure 3.7.2) from arbitrary namespaces including the target namespace.

```
<xs:element name="OptionalInputs" type="dss:AnyType"/>
<xs:element name="OptionalOutputs" type="dss:AnyType"/>
```

Figure 3.7.2: <dss:OptionalInputs> and <dss:OptionalOutputs>

The <dss:OptionalInputs> and the <dss:OptionalOutputs> are found in the <dss:RequestBaseType> in Figure 3.2.3 on page 55 and in the <dss:ResponseBaseType> respectively. Other extension points where dss:AnyType is used are <dss:InputDocuments>, <dss:SignatureObject>, <dss:KeySelector> and <dss:Timestamp>, to provide a free content model via the <dss:Other> element containers. These are locally defined as relevant to their parent elements.

3.7.1 Fixing the *OASIS-DSS Schema*

In *OASIS-DSS* WD30 page 10 extensibility was modeled differently and not conformant to *Schema's* unique particle attribution (UPA) rule. This rule requires that the content model of an element has to be unambiguous about which element declaration matches the currently validated element, without having

[12]The X in *XML* is not to be confused with *Schema's* type extension.

to inspect its contents. This means a *Schema* validator will be able to continue validating an element alone by its namespace and element name. An instance element can never be matched by more than one particle of a *Schema*.
The wild card definitions in *OASIS-DSS* WD30 page 10 would have matched any element in any namespace and are hence not usable in a choice unless distinguished from other particles therein.

A common extension strategy for a choice is to use the following expression in Figure 3.7.3 to discriminate by namespace. It allows any top level defined element as long as it is from another namespace than the target namespace.

```
<xs:any namespace="##other" processContents="lax"/>
```

Figure 3.7.3: Wild-card matching elements from outside the targetnamespace.

In case of <dss:InputDocuments> this would have been possible given that all other particles of the choice are in the target namespace.
The case is different though for <dss:SignatureObject> element declaration (Figure 3.3.11 on page 70) where this is not possible, because the choice contains elements from two namespaces. In this case a construct like the following would be necessary. *Schema* lacks a way to describe a namespace constraint by means of the complement of a set of namespaces.

```
<xs:any namespace="##not ##targetnamespace
   http://www.w3.org/2000/09/xmldsig#"/>
```

Figure 3.7.4: Wild-card do not allow to match the complement of sets of namespaces.

For <dss:KeySelector> and <dss:Timestamp> particles from a foreign namespace this issue applies as well as the only namespace negation allowed in *Schema* is the negation of the target namespace using `##other`. This limitation of *Schema* was obviously not anticipated by the other members of the *OASIS-DSS TC* and shows that *Schema* should maybe infer the complement set of namespaces automatically. *Schema* only supports sets of namespaces which is not useful however as one cannot predict which namespaces might be relevant in the future.
Although Figure 3.7.3 would have been possible for <dss:InputDocuments> and allowed profiles to plug-in additional payload containers peer to the others, the *OASIS-DSS TC* decided to follow only one strategy for extension points, namely <dss:Other>. <dss:Other> has its own content model and avoids this problem by introducing yet another container element. Trading off document's depth with clarity.

Chapter 4

Signing *XML*, weaknesses, solutions

The work on *OASIS-DSS* has shown that unconstrained *XML* signatures are in need for full fidelity round-trip[1] support for transmitting data objects such as binary and *XML* content. The most robust form of achieving this opaqueness is *base64 encoding*, however buying in the additional costs of encoding, decoding and increased space. Depending on the encoding of the transport protocol the size increases for instance to 133% for UTF-8 and 266% for UTF-16 [64] for binary data. Some of this increase would also occur for UTF-8 encoded in-lined *XML* just for trans-coding an UTF-8 document to UTF-16 even if carried in <dss:InlineXML>. The 66% increase is saved however. The *base64 encoding* further disallows direct access to contents of such data objects on a *SAX* level.
If *XML* containing *base64 encoding* is parsed to *DOM* such blobs are not deflated until they are actually used as binary data and are not held in the *DOM* tree any more. In a UTF-16 encoded Java™ `String` such a blob consumes 266% of its original size until it is garbage collected.

We have learned in the previous chapters however that a clean separation of transport protocol and payload is not only good practice, but is crucial to prevent *XMLDSIG* signatures from breaking due to to inherited context.
Filtering in-line *XMLDSIG* signatures or signed documents on a lower level such as *SAX* and *StAX* is possible, but gives away the architectural benefits of working on the level of tree model abstraction. A hybrid approach disallows exchanging services or implementations at the various layers. Hence *base64 encoding* was chosen to be the default for conveyed payload.

```
enveloped ::= '<xml' (S  Attribute)* S? '>' document '</xml>'

content   ::= CharData? ((enveloped|element|Reference|CDSect|PI|Comment)
                          CharData?)*
```

Figure 4.0.1: Rounding XML $\sqrt{2.0}$ [2]

[1] See later in section 5.3.1 on page 113.

[2] XML $\sqrt{2.0}$ is a public square where *XML* 2.0 should be rooted. This *BNF* is an amendment to the *XML* specification,

With the ideas in the following sections we will try to “think” out of the box and propose needed “minimal” changes to *XML* specifications to be able to have document level context separation and opaqueness while avoiding the costs of *base64 encoding*.

4.1 Change *XML*

The *prolog* cannot be child of an element, simple inheritable attributes such as *xml:lang* and *xml:space* get inherited as soon as one piece of *XML* is spliced into another one. The situation is even worse with *xml:base*, suddenly the “place of an entity” is not actually its place any more, at least not with respect to its contained references (or links). Namespace inheritance is another issue when one tries to envelope a piece of *XML* in a <ds:Object> to create an enveloping *XMLDSIG* signature or embed a document in some transport protocol like *OASIS-DSS*.

The approach to embed *XML* in a `CDATA` section suffers the problem that the embedded documents in turn cannot contain `CDATA` sections. To be opaquely transmitted, at least some parts need to be escaped. The *XPath* data model does not know `CDATA` sections. *XML* processing *APIs* like *DOM* or *XPath* are allowed to convert `CDATA` sections to escaped character sequences and merge them with adjacent text nodes.
The character data of a processing instruction seems to be a little more robust with respect to not being touched by tools, if processed at all. It potentially allows signed documents referred to by a <ds:Reference> with external *URI references* and no <ds:Transforms> (i.e. no implicit *C14n*) to be transmitted as long as their character encoding remains the same. This however could be perceived as a plain misuse of processing instructions and it is questionable whether tools would accept larger quantities of data as the character data of a processing instruction. Further many tools do not consider *processing instructions* as first class objects[3]. In Figure A.1.1 on page 120 these techniques can be tried out by copying the contents to a COTS[4] *XML* editor that supports syntax highlighting for `CDATA` sections and *processing instructions*.

XML is not closed under the operation embed.

Yet accepting all the idiosyncrasies of an *XML* document, an idea would be to extend *XML*, perhaps call it *XML* 1.4142 and let it roughly be what could eventually be the root of *XML* 2.0. It shall be closed under the operation of embedding one document into another one, i.e. the result shall be a well formed *XML* 1.4142 document again.
Separation of form and content [65] made *XML* revolutionary, however if used as a structured transport protocol it lacks the separation of payload from protocol. Allowing *XML* documents to contain *XML*

explained in the next sections.

[3]*Also, in new developments, all significant objects with any form of persistent identity should be ”first class objects” for which a URI exists. [65]*

[4]Commercial off-the-shelf

documents allows them to be passed by their true value preventing signature breakage and avoiding the additional costs *base64 encoding*.
Architecturally *URIs* can then be dereferenced and passed by value in an *XML* protocol, retaining human readability and increasing efficiency.

4.1.1 A Proposal for *XML* 1.4142

Many proposals for an *XML* 2.0 have been made in the past and Walsh [66] enumerates authors from Tim Bray, who drafted XML-SW [67] to Liam Quin and mentions people that have written essays, proposals to address the problems of *XML* and some suggest a successors of *XML*.
Walsh made an interesting proposal that the document production in *XML* should match the production of an external parsed entity [68]. An external parsed entity is the element's content production preceded by an optional text declaration[5]. Like a text document, without *prolog* having multiple document elements comprising a "markup forest"[6].

```
TextDecl       ::=     '<?xml' VersionInfo? EncodingDecl  S? '?>'

extParsedEnt   ::=      TextDecl? content
```

Figure 4.1.1: external parsed entity [6]

The disadvantage of this proposal is that the "TextDecl" is optional and hence external parsed entities are not delimited and surrounding text merges when one document is placed in another one.

This thesis in contrast to previously made proposals is driven by the need for round tripping signed documents without interfering with the payload. It also strives to achieve real opaqueness on a character level yet allowing *XML* to be the container as well as the containment. It assures that also the payload will not depend on surrounding context, which is an important architectural property.

```
document  ::= prolog  element  Misc*


element   ::= EmptyElemTag
            | STag content ETag

content   ::= CharData? ((element|Reference|CDSect|PI|Comment) CharData?)*
```

Figure 4.1.2: *XML*

The original definition of the "element" and "content" non-terminals can be found in Figure 4.1.2 and are obviously in the *XML* recommendation [6].

[5]A text declaration is an *xml declaration* lacking the `standalone` declaration.
[6]To stick to the analogy: the outer text nodes like bushes between the trees.

```
enveloped ::= '<xml' (S  Attribute)* S? '>' document '</xml>'

content   ::= CharData? ((enveloped|element|Reference|CDSect|PI|Comment)
                         CharData?)*
```

Figure 4.1.3: Required changes to XML[7]

Figure 4.1.3 summarizes the proposal by taking advantage of the fact that the name "xml" is reserved [6] in *XML*. It extends the content model of elements to be able to contain the non-terminal "enveloped" which in turn allows elements to include *XML* documents when they are delimited by an "xml" element. Whether such an element should be called "xml", "xml:xml", "xml:doc" or similar is not of importance.

By redefinition of the "content" non-terminal, documents would be allowed to reside in other documents. Despite of the encoding these shall share nothing with their host document, but the base *URI* of the containing element, essentially spawning a new parser.

Whether empty documents should be able to contain documents, meaning that `<xml>` can be child to `<xml>` or the file entity is out of the scope of this proposal, but may be a useful result for query languages wanting to report the empty document. Whether such should be well-formed *XML* 1.4142 is however questionable.

The attributes inside the start tag should remain reserved for future standardization purposes as the name "xml" is reserved now[8]. Such a change to the syntax of *XML* would allow a <dss:SignRequest> to look like in the following Figure 4.1.4. It would allow for character level fidelity round-trip support in the *OASIS-DSS* client server protocol.

In *Schema* simply an element declaration could be added potentially called `xs:xml`.

Given that documents can contain documents, path and query languages starting from context outside of such an internal document node within a document shall view it as a special document node where all context was lost and not being able to enter a document node unless doing so by a pseudo retrieval action.

This would make things that are now artificially complex in *XML* cleaner and easier and one would not have to change back and forth between *base64 encoding*, <dss:InlineXML>, *SOAP* with attachments or zip files for packaging *XML*. It would be a little bit like one of the most successful data structures in personal computing - the folder file concept - with the difference that the order would matter and between the files text would be allowed. Maybe that would be the first real markup system. Technologies such as Efficient XML Interchange (*EXI*) show that there is the possibility of efficiently encoding *XML*. Whether it could be efficiently mapped on a file-system, would be a topic for research.

[7]Analogously to tag definitions in *XML* [6] the end tag should be probably `'</xml' S? '>'`, but the author of this thesis never quite understood what the whitespace in the end tag is good for.

[8]Maybe attributes could indicate if a change in encoding will appear or this is the only true place for *xml:base*, content-length and other useful things now in *HTTP*. A change in encoding could make a document however unreadable, why such would be useful on the wire only.

```
<dss:SignRequest xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema">
  <dss:OptionalInputs>
    <dss:SignatureType>urn:ietf:rfc:3275</dss:SignatureType>
  </dss:OptionalInputs>
  <dss:InputDocuments>
    <dss:Document RefURI="http://www.example.org/foo.xhtml">
      <xml xml:base="http://www.example.org/foo.xhtml"
      ><?xml version="1.0" encoding="UTF-8"?>
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
      <html xmlns="http://www.w3.org/1999/xhtml">
        <head>
          <title>XHTML</title>
        </head>
        <body>
          <script type="text/javascript">
            <![CDATA[
            ... unescaped script content ...
            ]]>
          </script>
          <p id="same">Text</p>
        </body>
      </html>
      </xml>
    </dss:Document>
  </dss:InputDocuments>
</dss:SignRequest>
```

Figure 4.1.4: A simple example of a <dss:SignRequest>

4.1.2 Alternative proposal - *xml declaration*

An alternative to the proposal in the previous section would make the *xml declaration* `<?xml ...?>` an allowed syntactic element within an element. This element would then be the container for a complete *XML* document including the *prolog*. However, as mentioned, despite of its encoding such a document shall share nothing with its host document, but the base *URI* of the parent element.
This *xml declaration* would initiate a document child of an element and the document would be delimited by the end-tag of that element or an optional closing *xml declaration*. A closing *xml declaration* could be called enclosure[9]. This enclosure could have the form of a processing instruction `<?xml /?>`

[9]To avoid confusion with a closure.

4.1.3 Indention, Whitespace

Coming back from an enthusiastic introduction to the problems of signing markup. One of the best and at the same time most irritating features of *XML* is its indention and the concept of ignorable whitespace. *XML* is most well known for being an enabler to separate content from presentation and that is one of the main reasons which makes it so much better than *HTML*, but that does not seem to be true for *XML* itself. The *XML* specification until today lacks a simple dogmatic and clear statement of the form:

Every whitespace before and after a tag is purely for the readability of *XML* as such, even whitespace between a tag and the beginning of text or multiple whitespace at the line end before and after a line break should be considered indentation and hence not signed. Nevertheless processors shall leave it unchanged. This is true unless the *xml:space* attribute is set to “preserve” in which case whitespace is intended to be understood as carrying information and shall be signed.

Well isn’t this currently true for *XML*? No, but it should be[10].

A clear and common understanding about this would make *XML* securer and *XMLDSIG* a lot more flexible, because signatures would be more robust and thus better support actually signing the real information electronically (subsection 2.1.2). Less signatures would break and hence less decisions would have to be taken despite a broken signature, which could legally still be ok. It would further make what is discussed in the following sections redundant.

Whitespace, who cares?

By Eastlake [11] was pointed out that all white space with respect to *XMLDSIG* is considered significant and a poem is brought as an example. Whether a poem would ever be signed electronically is not discussed in this source, but it would seem appropriate for poem writers or their typesetters to escape or designate whitespace if it is significant. It may be a valid argument that in *XML* indention and whitespace is a means of structuring the markup and carries as such information. But just as with the poem it remains a very questionable use case for *XMLDSIG*. It is not contested that whitespace shall be preserved, but it shall not be signed.

The use case more important for *XMLDSIG* is pre-formatted text. Currently all content in *XML* is assumed to be pre-formatted with respect to *XMLDSIG* including structural elements and indention. So in reality the majority of signed content is just indented markup. An opt-in policy would have been more appropriate for *C14n* than a complex opt-out policy for removing whitespace via *XPath*-filter transforms in a <ds:Transforms> with a filter-step as in Figure 4.1.5.
XML offers an attribute called *xml:space* that can be set to `preserve` whitespace, if it really really matters. Thus allowing *C14n* to properly discriminate. *XPath*-filter transform can rather tediously also keep such space by applying Figure 4.1.6.

[10]Readers may note that whitespace even in this document matters, but what is really here is \smallskip, \medskip and \bigskip of LaTeX, and not the whitespace of the source code, or how DocBook’s LiteralLayout is indented outside its contents.

```
(//. | //@* | namespace::*)
  [not(self::text()) or not(normalize-space(self::text())="")]
```

Figure 4.1.5: An *XPath* expression to filter whitespace nodes.

```
(//. | //@* | namespace::*)[
  not(self::text()) or not(
      normalize-space(self::text())="" and
      not(ancestor-or-self::*/@xml:space[1]="preserve")
    )
  ]
```

Figure 4.1.6: An *XPath* expression to filter whitespace nodes, yet respecting *xml:space*.

Problems with signed base64 encoded text remain however. The only interoperable means for normalizing whitespace inside text nodes such as in *base64 encoding*, is the not very strongly supported and lately tarnished [40][69] optional *XSLT* transform[11]. For nodes consisting entirely of whitespace *XPath*-transform as shown above can be written. However to normalize space in *base64 encoding*, often appearing in the <ds:KeyInfo>'s <ds:X509Data> or in *XAdES* properties for embedding certificates in *XML*, is not possible. Here nodes need to be changed and not filtered and this requires *XSLT* as in Figure A.2.1 on page 121.
The *XMLDSIG* Base64 decoding transform offers an alternative. And from a first look it is not clear why for example *XAdES* in its latest version 1.3.2 [46] signs *base64 encoding* as is, ignoring the limitations of *base64 encoding* indention and does not make use of the *XMLDSIG's* Base64 decoding transform. One reason may be that *XMLDSIG* section 6.1 mentions it as "*Encoding*" and not as "*Decoding*" and only when reading further in section 6.6.2 one notes that this is actually a decoding. It is a useful mean to robustly sign *base64 encoding* embedded in *XML*. *XAdES* however concatenates and interleaves base64 encoded certificates with other readable content to digest it. One could argue that such data is only digested and mixing character data with binary data would not pose a real problem. This however makes the theoretical possibility of inspecting the `DigestInput` unpractical and hence violates *XMLDSIG's "See" What is Signed* (section 8.1.3 [29]). So the problem of normalizing the *base64 encoding* re-arises and *XAdES* simply does not normalize it, having to count on that such content is not to be touched. This however may hinder the spread of technologies such as *XMLDSIG* and *XAdES* because they can not be used safely with common tools.

Interestingly there exists no value for *xml:space* that allows applications to declare that whitespace like in *base64 encoding* is not at all of significance, not even single ones, not even before or after line-breaks including the line-breaks themselves, irrespective of whether such whitespace is preserved. All *XML* defines is: If *xml:space* is set to default, that means a parser must pass all characters to an application,

[11]Selections in *XMLDSIG* unfortunately do not allow for this granularity and only optional transforms such as *XSLT* can be used. We also remember that *XPointers*are still a working draft.

however the application can then do whatever it wants to that whitespace. And "preserve" causes the preservation of all whitespace.

So *xml:space* could be a means to achieve robustness at least for plain whitespace nodes[12].

This looks like a promising solution, but has the limitation that it will only work for non-validated well formed *XML* and valid *XML* whose grammar has anticipated the use of *xml:space*.

Schema and *DTD* add yet another way to discriminate whitespace by distinguishing between what is called *mixed content* from element content what will be discussed in the next section.

Validation for whitespace removal

Pure *XML* that has not been validated against some *DTD* or *Schema* does not make a distinction between whitespace that is used for indention of *XML* or the one that is used in a potentially pre-formatted text that is actually significant.

DTD and *Schema* introduce the notion of *mixed content* vs. plain[13] element content. The latter's intent is to flag all whitespace that exists between elements of such a parent element as "ignorable" and this very information is part of the Post Schema Validation Infoset (*PSVI*)[14].

Plain element content is when a valid element may only contain elements and no text nodes.

The opposite are elements having a *simple type* (i.e. an element contains only text, comments and processing instructions) or *mixed content*. In *Schema* this is indicated by the `mixed` attribute in `xs:complexType`. A *DTD* formulates *mixed content* like in expression Figure 4.1.7 by the union of `#PCDATA` with elements and must be declared first. In *mixed content* as specified by *DTD* only the kind of elements that may appear are constrained but not in what order or how often they appear.

```
<!ELEMENT e (#PCDATA|a|b|c)*>
```

Figure 4.1.7: Mixed content in *DTDs*

Schema further replaces default whitespace handling with two other options to provide a partial match for the whitespace normalization rules that apply to attribute values [23]. It should be added that also simple types can be controlled via the `whiteSpace` constraining facet, to preserve, replace or collapse them.

[12]Assuming subsequent <ds:Transforms> do not depend on the existence of these nodes in the underlying document's tree model or a canonicalization is performed immediately afterwards yielding a new document.

[13]The word plain is added in this thesis to distinguish element-content from the more general meaning of the contents of an element.

[14]`http://www.w3.org/TR/xml-infoset/#infoitem.character`

4.2 *XMLDSIG* and *C14n*

Having criticized *XML*, the whitespace normalization with respect to *Schema* validation topic seamlessly switches into the problem domain of *XMLDSIG*. In this section *ScC14n* is referred to as a rich source for having identified problems surrounding *XML* canonicalization. This is however not an endorsement of *ScC14n*. Its use is often not justified and the trade-off between canonicalizing more or less has to be done on an application basis. Things like whitespace or data-type normalization can not be performed separate from other normalizations because *ScC14n* does not give implementations the chance to only implement certain subsets of its specification. Related work has been performed by Geuer-Pollmann, who edited results of a workshop [70] where several issues in *XMLDSIG* were identified, they are in German and in keyword style. They have however not made their way back to *W3C*. Things additionally identified during the work on *OASIS-DSS* are mentioned hereafter.

For whitespace, namespace, attribute value and other normalization as described in *ScC14n* to work in an interoperable fashion, signers and verifiers need to perform *Schema* assessment equally, for which *XMLDSIG* once had a *Schema* validation transform [71][15] [11]. Until today it is only a working draft and hence not supported by implementers of *XMLDSIG*. *Schema* Instance can be used to associate a *Schema* with a document. *XMLDSIG* does not specify that the presence of hints[16] for document validation against a *Schema* or *DTD* have to be respected and validation to be performed. *XMLDSIG* only warns to use *Schema* consistently, but does not provide any means to enforce this [27][29]:

> Note, if the Signature includes same-document references, [XML] or [XML-schema] validation of the document might introduce changes that break the signature. Consequently, applications should be careful to consistently process the document or refrain from using external contributions (e.g., defaults and entities).

For increased interoperability *XMLDSIG* should specify a default. For example that validation hints if present in the document must be respected and processed.
Further a very simple parameter for dereferencing and parsing could clarify the situation. Current markup however does not allow to parametrize parsing. Hence a <ds:Transform> would be required and have the sole purpose to describe if parsing should be performed with validation and was performed when parsing the resource during signing. It could have one element parameter indicating one of the three options for a <ds:Reference>:

- `DoNotValidate` - just process as well formed *XML* document.
- `ValidateWithHintsInDocument` - use hints in the document.
- `Validate` - validate with supplied hints, ignoring potential hints in the document.

[15] `http://www.w3.org/Signature/Drafts/xmldsig-transform-xml-validation.html`
[16] *DOCTYPE* in subsection 2.1.1 on page 5)
`xsi:schemaLocation` and/or `xsi:noNamespaceSchemaLocation` in subsection 2.2.2.1 on page 13

This could be modeled in a *Schema* construct as follows in Figure 4.2.1. The first two options would reflect the current situation without requiring a verifying application to guess or derive a *Schema* from the application context. The third would provide an additional mean to supply schema information overriding the hints in the referred document. The third variant is not discussed in detail and a type declaration example how this could be done is in section A.4 on page 124.

```
<xs:complexType name="ValidationParameterType">
  <xs:choice>
    <xs:element name="DoNotValidate"/>
    <xs:element name="ValidateWithHintsInDocument"/>
    <xs:element name="Validate" type="vp:ValidateType"/>
  </xs:choice>
</xs:complexType>
```

Figure 4.2.1: ValidationParameterType

A signer could then deliberately either decide to ignore schema validation and prune information added by *Schema* or alternatively explicitly require a signer to validate against a *Schema*. Having such a discrimination would further enable system architects to locate signature validation either below or next to the *Schema* validation and an application logic.
This is somewhat similar to the <dss:Schemas> referred to by `SchemaRefs`. *Schema* validation is not only required for canonicalizing whitespace, but also to refer to external subresources by means of their fragment. Having a *Schema* validation transform would enable the use *ScC14n*, which itself does not provide any means to supply a schema, primarily because it also needs to be able to work on node-sets.

4.2.1 *C14n*, remove whitespace by default?

Schema assessment and *ScC14n* as the highest form of canonicalization are however very expensive. *C14n* as the current default was not defined to be robust against changes in whitespace, re-indented blocks of text or allowed whitespace in simple types such as *base64 encoding*.
XML users however have certain expectations concerning their freedom to indent and re-indent[70], when authoring or generating *XML*. If this freedom is to be maintained in *XMLDSIG*, signatures need to be made robust against such changes yet avoiding many of the costs of *ScC14n*.
So do we have to live with the following dogma, also for signing and verifying content?

> Whitespaces do matter in XML. Adding a whitespace for indention changes an *XML* document and it is not the same document anymore [9].

By default whitespace in an *XML* document is significant and should be preserved unless the parser is by means of a *DTD* or *Schema* instructed to ignore it. Nevertheless, inconsistent whitespace handling is one of the major sources for *XMLDSIG* verification and interoperability issues. Removing whitespace using <ds:Transforms> preserves the original document yet assuring a normalized form to be signed. *XPath*-transforms as the ones described previously in this chapter (Figure 4.1.6 on page 83) try to avoid as much navigation as possible and could be easily implemented without employing a full *XPath*-implementation. This however helps only for the <ds:References>, but is not applicable for the <ds:SignedInfo>. There only a <ds:CanonicalizationMethod> is allowed and as *C14n* has decided to preserve all whitespace (subsection 2.5.2) there seems to be a deadlock situation forcing *XMLDSIG* application designer to a very inflexible: "Do not touch signed documents at all".

Which is why there is the need for standardization in that area.

Technologies like *XML* binding frameworks or web service enabling technologies such as Axis (*Axis*) or Glassfish[17] are often used in applications. When these applications also use *XMLDSIG* signatures, such tools often break the signatures due to their differences in whitespace handling. Which is with respect to the default whitespace processing violating the *XML* specification.

> An XML processor MUST always pass all characters in a document that are not markup through to the application. (section 2.10 [6])

Sometimes even *XSLT* implementations vary with respect to their whitespace handling.

For many applications whitespace is insignificant and for *XMLDSIG* applications it almost always is, or at least it should be given their slightly different interpretations in various tools.
The complexity of *XPath*-expressions needed to formulate an *XPath*-transform often keeps users from consciously removing whitespace before they sign documents and reflect this in the <ds:Transforms> of the <ds:Reference> in question.

4.2.2 Making signatures robust against changes in whitespace is crucial.

So after all it seem necessary appropriate to live with the whitespace dogma and the principle that signed documents should not be touched. This shall not be contested, but extended. In the last few pages it was shown that the treatment of whitespace is specified, but not in a manner that is easily interpretable by the average user.

Making signatures robust against changes however, respects that instances of *XMLDSIG* applications and its deviates are required to follow the web architecture [72] specifically *tolerance ... the life and breath of Internet* [65].

> "Be liberal in what you require but conservative in what you do" [73].

Translated to *XMLDSIG* this means: "Refer only to what is necessary, and canonicalize as much as possible by default!"

[17] An open source application server project for the Java™ enterprise edition.

Saying something is application dependant or expensive is a mere excuse of engineers not trying hard to figure out to make it robust and efficient. Principles for designers of user agents such as browsers or *XMLDSIG* applications have to be proxy for their end users. *OASIS-DSS* allows them to do this centrally in office environments, but such should apply for decentralized application developers as well:

- "Signer, should be conservative in what they consider as being the Information they want to have secured."
- "Intermediaries, are invited to process signatures with whatever tools they find appropriate. Be conservative in what you have to touch for processing, especially do not touch signed documents and use opaque containers (subsection 3.2.3 on page 57). If yet available `<xml> ... </xml>` (subsection 4.1.1 on page 79)."
- "Intermediaries and verifiers, do not touch what was meant to be signed, and hence has been signed or the signature breaks."
- "Verifiers, only what is signed (i.e. `DigestInput`) should be shown as signed or processed as signed."[18]

Balancing the trade-off between robustness, efficiency and simplicity can not mean only to resign and hide behind a "Do not touch signed documents at all" principle. This will hinder the spreading, processing and passing on of signed content, yes signed information entities that can be trusted, across the Internet.

4.2.3 Broken Signatures

Ideally a signature should only break if at least one of the referred data objects has been changed in its information contents. The creation of *XMLDSIG* signatures has a number of pitfalls. It is quite easy to accidentally produce "false negatives". After allowed processing has been applied that broke the signature, it could nevertheless be valid and verify if for instance the correct *Schema* would be used for parsing. Problems like attribute and namespace inheritance, data-type normalization, surrounding context such as a *SOAP* protocol or different handling of whitespace, may cause the signature verification to fail.
XML given, all its flexibility begs the need of a normal-form of canonicalization, that can in many cases be application dependant and realized by custom transforms. Often the current normal-form (*C14n11*) is perceived as not performing well in terms of speed and normalization capability. This thesis argues that it is not normalizing enough.

It is after all also a matter of fact that "false positives" destroy trust, but "false negatives" do so as well.

[18]A modification of [29]. Note further that if no <ds:Transforms> or only trusted <ds:Transforms> are used there might be equivalence between `DereferencedData` and `DigestInput`.)

The achievable robustness of a signature can only be exploited to its full potential on signing. When a signature has been created and sent off the verification is merely reproducing what the signer prescribed in the chain of transforms if present. So signing documents centrally in an office environment will allow to make the right choices once for all users. A *OASIS-DSS* server can be a means to take this burden of individuals and is however on the other hand side not preventing decentralized verification of signed document, given that all data objects are made available.

Key discovery for the various cryptographic signature schemes from DSA to RSA over key distribution concepts such as PGP, SPKI, *PKI* (X.509 encoded in ASN.1 BER, DER, CER) to the proper and secure choice of hash functions, and so forth with trustworthy <ds:Transforms> eventually only showing "what is signed", constitute the complexities and choices that need to be taken off a client and are better manageable centrally on just one system for a group of users.

If everything is done in the right way however signatures offer End-to-End integrity of data and authenticity. They can be stored and as in the case of *XAdES* remain processable and secure over log periods of time.

Their visual representation must be trustworthy as well, so "What You See Is What You Sign" [8].

4.2.4 Proper use of XSLT in XMLDSIG

Hill pointed out the the order in which *XMLDSIG* signatures are processed during signing should be reversed on verification [40] [69]. This means that first a key should be selected and verified as trusted. The the reference processing should only be executed after the <ds:SignatureValue> and the <ds:SignedInfo> have been verified. Hence at least the authenticity of the signature value and the integrity of the <ds:SignedInfo> would have been established.

Nevertheless it may be easy for an adversary to acquire a good existing signature. Let *XSLT* be further be included by `xsl:include` or `xsl:import` (subsection 2.3.6 on page 33) in one of the <ds:Reference>'s <ds:Transforms>. Then it has not been authenticated together with the verification of the <ds:SignedInfo>, because only its location or the *URI reference* pointing to the imported transform has been secured. Thus an adversary can supply a transform of arbitrary choice on verification assuming that it is either a co-located by a *relative URI reference* or the adversary has control over the referred location. Which is especially critical for *OASIS-DSS*.

Hence additionally to what Hill suggested [40] *XSLT* transforms external to the <ds:SignedInfo> should only be executed on verification, if they have been covered in a <ds:Reference> (Rx) computed before applied in another <ds:Reference>'s (Ry) <ds:Transforms>. Preferably Rx's digest value should also be compared against a "known good value" avoiding attacks by authenticated signers on a *OASIS-DSS* sign request.
The data stream to acquire the *XSLT* transform secured by Rx should be the Rx's `DigestInput` to make sure the secured transform will be performed.

4.2.5 Enveloping legacy XML

As legacy *XML*, which has no default namespace declared, would inherit the default namespace of any surrounding namespace scope, legacy signatures when embedded into a <ds:Object> should clear the default namespace using `xmlns=""`[19].

```
...
<dss:Inputdocuments>
  ...
  <ds:Object xmlns="">
    <legacyelement>
    </legacyelement>
  </ds:Object>
  ...
<dss:Inputdocuments>
...
```

Figure 4.2.2: Enveloping legacy *XML* with *Exc-C14n*

This (Figure 4.2.2) is sufficient when signed with *Exc-C14n* if used with either implicit *C14n* or explicit *C14n* or explicit *C14n11* the inherited namespaces will be rendered. In *XMLNS 1.1* for *XML 1.1* namespace undeclarations are allowed [74] enabling namespace context separation as shown in Figure 4.2.3. Nevertheless as mentioned in Figure 2.1.1 on page 7 the *XPath* data model is not well defined for *XML 1.1*.

```
...
<dss:Inputdocuments>
  ...
  <ds:Object xmlns="" xmlns:ds="" xmlns:dss="">
    <legacyelement>
    </legacyelement>
  </ds:Object>
  ...
<dss:Inputdocuments>
...
```

Figure 4.2.3: Enveloping legacy *XML* with *C14n* and *XML 1.1*

Hence the only way in which legacy *XML* can be currently enveloped in a <ds:Object> without inheriting any namespaces is by means of re-declaring the default namespace and eventually undeclaring it in a separator element as shown in Figure 4.2.4.

Similar considerations also apply for non legacy *XML* which is to embedded in a <ds:Object> using *C14n* or *C14n11*. This section is hence yet another supporting argument for the need of being able to opaquely envelope *XML* in *XML*.

[19] 200504/msg00048

```
...
<Inputdocuments xmlns="urn:oasis:names:tc:dss:1.0:core:schema">
  ...
  <Object xmlns="http://www.w3.org/2000/09/xmldsig#"">
    <separator xmlns="">
      <legacyelement>
      </legacyelement>
    <separator
  </ds:Object>
  ...
<Inputdocuments>
...
```

Figure 4.2.4: Enveloping legacy *XML* with *C14n* and *XML*

4.2.6 Wrapping Attacks - merely neglecting “See” What is Signed?

“See” What is Signed (section 8.1.3 [29]) is often not correctly implemented and has lead to what McIntosh calls “Wrapping attacks” [75]. There multiple data objects having the same value for an `xs:ID` are in the same *XML* document or similar ambiguity exists causing an *XMLDSIG* implementation to dereference and secure other data, than will be processed by the application (see also Figure 2, page 3 given in [76]).
Wrapping attacks can appear, when *XMLDSIG’s* reference processing is used in “not-location-sensitive” manner and the application uses a mechanism different than the one used by the <ds:Reference>. Despite warnings in the specification [29], various implementations and standards operate on data allegedly referenced by a <ds:Reference>. The main reason for this is that different referencing mechanisms are used by the application and *XMLDSIG*.
In the case of <ds:Reference>s without <ds:Transforms> or only simple trusted transforms[20] the assumption holds, that the `DereferencedData`, despite filtered nodes, equates to the `DigestInput`. In this case the `URIDereferencer` can be overridden to either use the same mechanisms as the application for locating data objects and passing them directly to the application to operate on.
In the case of more complex <ds:Transforms> one should only operate on the `DigestInput`, which however contains the already canonicalized representation of the `DereferencedData`. The *JSR105 API* does not provide any means to access the pre-canonicalization node-set, which could be used either directly as an iterator to process the secured parts of a request or by object comparison the secured nodes can be selected as passed by on normal processing eventually.
In ([76] section 3.1) Gajek et al. propose a ”Strict Filtering approach”, which is from a first look not different to what the standard says in (section 8.1.3 [29]) already:

> [...] automated mechanism that trust the validity of a transformed document on the basis of a valid signature should operate over the data that was transformed (including canonicalization) and signed, not the original pre-transformed data. [...][29]

[20]E.g. such performing only whitespace normalizations

We hence call it the "See What is Signed approach".

Such an approach, operating on the canonicalized data, requires that the data has to be self contained. If the data is self contained however it is unlikely to be location dependent, which contradicts the pre-condition of a "wrapping attack" for which the location is essential in that an application that locates a different data object than the <ds:Reference>. McIntosh even claims [75]:

> In practice we can think of no realistic examples of purely context independent semantics.

The statement likely has to be seen in a web services context, as opposed to for instance enveloping signatures' <ds:Object> contents. Such are very likely context independent. Similarly for signatures, whose visual appearance is of significance as with such created by an *XSLT* transform for user agent display in a trusted viewer. Here the precondition is that the data and its derivative is self contained.

The "See What is Signed approach" unfortunately is not respected very often for automated processes. The data is often seen as pseudo self contained, mostly referred to by a short hand *XPointer* via an attributes of type `xs:ID` and a <ds:Reference> without additional transform. Such is also the most common use in web service frameworks, buying in the disadvantages mentioned in [76] and [75].

Example 6 in [75] proves that the use of *XPath*-filters is too complex as even security engineers do not get them right. This current expressions used there would one of the two things: If there exists an element as identified by the path the whole document including the signature itself (without signature value, as did not exist) is signed, or if this element does not exist the hash of nothing would be computed.

```
wrong XPath-filtering:
  ...
    <ds:XPath>
      /soap:Envelope/soap:Header/wsa:ReplyTo
    </ds:XPath>
  ...
right XPath-filtering:
  ...
    <ds:XPath>
      self::node() = /soap:Envelope/soap:Header/wsa:ReplyTo
    </ds:XPath>
  ...
as this is equal to the XPath:
  ...
  (//. | //@* | //namespace::*)
     [self::node() = /soap:Envelope/soap:Header/wsa:ReplyTo]
  ...
```

Figure 4.2.5: <ds:XPath> confusion.

The signatures can never verify again as the signature value will always be in the computation of the verification, which cannot have existed before signing. In short Examples 6 - 10 in [75] have a bug.

Nevertheless the general idea of locating the same objects as the ones being processed is right. What the authors wanted to express is that the same location mechanisms used in the application shall also be used in the signature by means of *XPath*-filtering. The easiest mean to do this however would be the *XPath* subset of *XPointers*.

```
...
  <ds:Reference URI="#xpointer(/soap:Envelope/soap:Header/wsa:ReplyTo)">
    ...
  </ds:Reference>
...
```

Figure 4.2.6: *XPointers* are simpler.

[76] further proposes to return location hints, which could be equally achieved by referencing the relevant parts in a location sensitive manner securely upfront as already suggested by [75].

This thesis claims that so called “Wrapping Attacks” are not a problem of the *XMLDSIG* standard, but of the used *APIs*. Hence we propose to extend the *JSR105 API* about an access to the pre-canonicalization node-set to allow implementations to operate on the actually canonicalized content. Further a method determining if such is is still followed by <ds:Transform>s but a final canonicalization of some sort.

4.2.7 *C14n* and the *XPath* Data Model

One of the current bottlenecks in *XMLDSIG* is *C14n* [77]. The reason for this is that the namespace fix-up in the *C14n* algorithm depends on the fact whether a namespace-node is in the node-set [52] [53]:

> **Namespace Nodes** - A namespace node N is ignored if the nearest ancestor element of the node's parent element [O] that is in the node-set and **has a namespace node in the node-set** with the same local name and value as N. Otherwise, process the namespace [...]

In tedious cases this can lead to documents that can only be canonicalized if it is known whether the ancestor (N) of the namespace node owner (O) AND N's namespace node have been selected to be in the node set or not.
Interestingly a situation can appear where N's parents namespace node will be rendered just as well as O's namespace node in question. Namely when N's namespace node is not in the node-set and N's parent does not have the same situation as O looking upwards the ancestor axis or it is the root node.
Taking the example from the example 3.8 from *C14n11* [53] in Figure 4.2.7:
Applying the following document subset expressions in Figure 4.2.8.

- . = parent::*/namespace::*
  the current node lies in its parents namespace axis and is hence a namespace node

```
<!DOCTYPE doc [
<!ATTLIST e2 xml:space (default|preserve) 'preserve'>
<!ATTLIST e3 id ID #IMPLIED>
]>
<doc xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org"
   xml:base="something/else">
   <e1>
      <e2 xmlns="" xml:id="abc" xml:base="bar/">
         <e3 id="E3" xml:base="foo"/>
      </e2>
   </e1>
</doc>
```

Figure 4.2.7: Example 3.8 from *C14n11*.

- parent::*/*
  the current nodes parent has children
- parent::*/parent::*
  the current nodes has a grand parent

```
  (//. | //@* | //namespace::*)
    [not(. = parent::*/namespace::* and parent::*/* and
       parent::*/parent::*)]
```

Figure 4.2.8: Document subset expression to show that *C14n11* is too complex.

Taken together and negated this causes in the current example all namespace nodes but those of <doc> and <e3> to be filtered away. So there is a gap consisting of <e1> and <e2> where the namespaces have been "undeclared"[21]. What this causes however is that the namespace for the namespace declaration `xmlns:w3` is rendered in <doc> and <e3> as stipulated before.

```
<doc xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org"
   xml:base="something/else">
   <e1 xmlns="">
      <e2 xml:base="bar/" xml:id="abc" xml:space="preserve">
         <e3 xmlns:w3c="http://www.w3.org" id="E3" xml:base="foo"></e3>
      </e2>
   </e1>
</doc>
```

Figure 4.2.9: Document subset expression to show that *C14n* is too complex.

Recalling now from subsection 2.3.3 on page 18 that namespace declarations in the *XPath* data model are not normal attributes in contrast to *DOM*. In the *XPath* data model they are all spread to their descendants. More precisely one could say they are accessible via the namespace axis from all descendant

[21]Such could only be represented in *XML 1.1*.

nodes of the element bearing the namespace declaration. They are “always there”, which however does not mean that they have been selected to be in the node-set and *C14n’s* algorithm prevents us from being ignorant about their existence in the node-set.

The currently used standard *XPath* implementation such as *Xalan* for Java™ requires the nodes to be distributed across the document. This however consumes large amounts of memory and for large documents significantly slows down the processing as the namespace distribution is quite expensive.

It is a topic for future research to see whether it is possible in current or future versions of *Xalan* to get the same node returned for all elements on which its scope falls. This would mean to have a namespace node multiple times in the node-set, appearing on its descendant-or-self axis unless redeclared and not copies thereof distributed across the node-set. This would significantly aid to lower memory consumption and increase performance, especially for large documents.

If this is not possible however, a simplification of *C14n* would be the approach that has to be considered for standardization.

4.2.8 Should fragment URI references strip comments?

In Figure 2.3.15 on page 26 it is mentioned that comments are stripped depending on the form of <ds:Reference>’s `URI` as being either an external *URI reference* or a *same-document reference*. *XMLDSIG* however does not clarify whether *comments* are considered to be removed at parse time or whether they are considered to be in the data model and just removed from the node-set. In the latter case, expressions in the <ds:Transforms> could still depend on the *comments* and *XPath*-filter 2.0 [49] could even re-import them.

One can argue here that too much functionality is implicitly put into the special values of *URIs*, hence making the standard unnecessary complex. It is not understandable following the principle of least surprise, why the full *XPointer* `#xpointer('/')` should dereference a different resource than the one returned by URI=`""`.

4.3 *OASIS-DSS*

In chapter 3 *OASIS-DSS* has been introduced and an analysis of the protocol has been provided. Some points however which will follow in the next sections are better located in this chapter as their level of detail is higher.

4.3.1 Context Free Extraction and Opaqueness

Signature verification errors can occur due to inherited namespace declarations around the signature, other than those when the signature was generated. *Exc-C14n* was devised to address this, however additional problems can occur.

If expressions like for example *XPath*-Expressions inside *XPath*-Filters (section 6.6.3 [29]), *XPath*-Filters 2.0 [49], Canonicalization (explicit, implicit *C14n*) or *XSLT* (section 6.6.5) are used in the chain of transforms, so that they refer to parts of the transport protocol, signatures are sensitive to their context. This may either happen accidentally, in which case we talk about "spurious verification errors", or on purpose, then we talk about "chosen context verification errors".
Such transformations are not so uncommon as they simply walk up the *XPath* ancestor-axis or refer to absolute parts, that may be changed by processing, and include or exclude elements depending on the state of the transport protocol. *XPath* expressions that use the namespace axis (`namespace::*`), *xml:base*, *xml:space*, *xml:lang* are also affected by the surrounding inherited context. Similarly for *XPath*-Filters 2.0[22] and *XSLT*.
In short all non opaquely conveyed payload in *OASIS-DSS* needs some form of context free extraction, before it is processed.
Such payload elements in *OASIS-DSS* are:

- <dss:InlineXML> - not allowed to be of *mixed content* and requires *Exc-C14n* (or equivalent in memory processing).
- <dss:SignatureObject>'s <ds:Siganture>
- and if copied on an *Infoset* level:
  - <ds:Transforms>
  - <ds:KeyInfo>
  - <ds:DigestMethod>
  - <ds:DigestValue>

Avoiding this would enable the creation of signatures being valid in the context of one system, but not valid in the context of another. If they are used in a way so that they may also refer to parts of the surrounding context, for example a transport protocol such as *OASIS-DSS*, then the output will be different depending on whether the document is inside that context or not. This can result in verification errors.
Hence somebody may easily commit to create a valid signature in system A, if he or she has the possibility to choose or influence a system B with another context in which the signature is to be verified (200608/msg00012). The person may still be legally bound to the commitment, but the advantage of having a verifiable signature as evidence may be gone in system B. If system A ceased to exist debugging a signature in system B can become very complex.
The means identified by the *OASIS-DSS TC* to assure consistent behavior is "context free extraction" of signed in-line XML content as follows for <dss:InlineXML>.

[22]It is also questionable whether *XPath*-filter 2.0 [49] being allowed to re-include nodes that have been previously filtered in the <ds:Transforms> is good design.

4.3.1.1 Context Free Extraction from <dss:InlineXML>

<dss:InlineXML> is a means to transport documents inside *XML* opaquely in the lack of *XML* being closed under the operation embed as discussed at the beginning of this chapter in section 4.1 on page 78. It effectively is the workaround taken by *OASIS-DSS* to opaquely transport *XML* in *XML*. It avoids the need to specify extraction at an implementation level such as *SAX* or *DOM*.
In *OASIS-DSS* Working Draft 30 this element was called <dss:XMLData>[23] and intended to transport arbitrary intermediate results of <ds:Transforms>. How fragmented node-sets would be transmitted and eventually canonicalized was unspecified as well as if and how the serialization at the client-side (cf. subsection 2.5.3.1 on page 46) and the parsing at the server-side[24] would work[25]. Problems like context independence discussed in the previous section also applied.

Hence the *OASIS-DSS TC* decided to move the functionality sought by <dss:XMLData> into the <dss:TransformedData>, which may be used in combination with <dss:SignedReference> (cf. client-side splicing subsection 3.3.5.1 on page 66 and subsection 3.3.6 on page 66).

<dss:InlineXML> was retained for in-line transmission of complete documents. Its has the limitation implied however by using *Exc-C14n* for extraction, not to use *DOCTYPE*, *DTDs* or *QNames* in content. The *OASIS-DSS TC* should specify an additional parameter or optional input for *Exc-C14n* to mitigate the latter limitation (cf. `InclusiveNamespacePrefixList`).

```
<xs:complexType name="InlineXMLType">
  <xs:sequence>
    <xs:any processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="ignorePIs" type="xs:boolean" use="optional"
     default="true"/>
  <xs:attribute name="ignoreComments" type="xs:boolean" use="optional"
     default="true"/>
</xs:complexType>
```

Figure 4.3.1: dss:InlineXMLType used in dss:DocumentType

<dss:InlineXML> defined locally inside dss:DocumentType (see Figure 3.2.7 on page 57) is of type dss:InlineXMLType. Unlike optional inputs, optional outputs or extension points dss:InlineXMLType has not been specified using dss:AnyType.
dss:AnyType has a *mixed content* type. When looking at the *prolog* and Figure A.5.1 on page 124 one can see that character content, before and after the *document element* is only allowed to contain ignorable whitespace. dss:InlineXMLType is not of *mixed content* and contains only one element of

[23]In WD30 page 15, <dss:InlineXML> was called <dss:XMLData>

[24]Node-sets are not necessarily well formed *XML* and may carry information in their *PSVI*.

[25]Ambiguities concerning namespaces and the signing of `InlineXML`, if *C14n* and not *Exc-C14n* was used in WD30 page 15 section 3.3 step 1a. The same was true for 3.4 1a, if a DOM based parser is used extract the info from inside <dss:XMLData>.

type `xs:any`, which assures its content is well-formed *XML* by itself lacking only the *xml declaration*. Assuming it was serialized it could be parsed again yielding a proper *XML* document, however not supporting a Document Type Declaration (*DOCTYPE*) and hence lacking support for external and local *DTDs*.

DOM does not provide clear means for context free extraction, yet keeping used namespace-declarations (200510/msg00003) and loosing inherited namespaces. Specifying context free extraction, by means of *DOM* would further have bound *OASIS-DSS* to *DOM*.

The *OASIS-DSS TC* decided hence to specify "context free extractions" by means of *Exc-C14n*. Context free extraction [62] section 3.3.2 mandates the use of *Exc-C14n*.

OASIS-DSS implementations using *DOM* can still use in memory extraction of <dss:InlineXML>. This can be achieved by using the *DOM* methods "importNode"[26] or "adoptNode"[27] and the algorithm defined in section 3.1 "Constrained Implementation" of *Exc-C14n* [55].

<dss:InlineXML> has two attributes `ignorePIs` and `ignoreComments`. They Indicate respectively, if processing instructions or comments MAY be ignored. They shall reflect that *XML* processors such as binding frameworks often do not pass *processing instructions* to the application [78][28] in violation to what is specified in *XML* [6].

> [...] *processing instructions* are not part of the document's character data, but MUST be passed through to the application. [6]

For *comments XML* specifies only that processors may make them available for applications [6]. Things will hopefully be better in newer versions of binding frameworks.

> When synchronizing changes to *JAXB* view back to related *XML Infoset*preserving view, every effort is made to preserve XML concepts that are not bound to *JAXB* objects, such as XML *Infoset*comments, processing instructions, namespace prefix mappings, etc [*sic!*] [79]

[26]"`http://www.w3.org/TR/DOM-Level-3-Core/core.html#Core-Document-importNode`"
[27]"`http://www.w3.org/TR/DOM-Level-3-Core/core.html#Document3-adoptNode`"
[28]"`http://markmail.org/message/mkpwr5fiwb3rzaji`"

4.4 Conclusions

- XML is not closed under the operation embed.
  - It should be extended: The content model of element should be able to contain the non-terminal "enveloped" which in turn allows elements to include *XML* documents when they are delimited by an "xml" element.

    ```
    enveloped ::= '<xml' (S  Attribute)* S? '>' document '</xml>'

    content   ::= CharData? ((enveloped|element|Reference|CDSect|PI|
                             Comment) CharData?)*
    ```
  - Alternatively make the *xml declaration* `<?xml ...?>` an allowed syntactic element within an element, followed by the prolog. A closing *xml declaration* called enclosure of the form of a processing instruction `<?xml /?>` should optionally delimit a document.
  - Otherwise the additional costs of *base64 encoding* remain inherent to transporting *XML* in *XML* in the general case.
  - Workarounds like <dss:InlineXML> work for a subset of *XML* documents and have their limitations. They not even carry so common *XML* formats like *XHTML*.
- A simple dogmatic and clear statement in the *XML* standard of the form ...

  Every whitespace before and after a tag is purely for the readability of *XML* as such, even whitespace between a tag and the beginning of text or multiple whitespace at the line end before and after a line break should be considered indentation, as such *XML* processors MUST preserve it however. Unless otherwise indicated by some form of Schema language or by means of *xml:space*=`"preserve"` it does not carry information and hence SHOULD not be signed.

  ... would allow to retain *XML's* user expectation about indenting and re-indenting *XML* for signed documents.
- Making signatures robust against changes in whitespace is crucial, and *XMLDSIG* lacks a means to do this at the <ds:SignedInfo> level.
- "False positives" destroy trust, but "false negatives" do so as well.
- We argued that "Wrapping Attacks" are not a problem of the *XMLDSIG* standard, but of the used *APIs* and better access to the pre-canonicalization node-set "`CanonInput`" is required.
- "`CanonInput`" is necessary means to perform a "See What is Signed approach".
- Additionally to reverting the order of *XMLDSIG* reference processing model for verifying (Hill [40]), external *XSLT* transforms should be covered by a previous <ds:Reference> and the digest value should be preferably compared against a "known good value" before applying the *XSLT* transform retrieved from the previous <ds:Reference>'s `CanonInput`.

Chapter 5

OASIS-DSS prototype library

For this thesis the manner in which the necessary components are organized and integrated for *OASIS-DSS* has been studied to derive an architecture for a *OASIS-DSS* software library in Java™. A design has been made and implemented by means of evolutionary prototyping and yielded DssXp. Digital Signature Services XML processor (*DssXp*) is the library created by the author of this thesis. It has been successfully used and extended by Zwattendorfer and Zefferer for a reference implementation and a demonstration project [80].

5.1 Architecture

Architecture - Software architecture for a system is the structure or structures of the system, which consist of elements and their externally visible properties, and the relationships among them. [81]

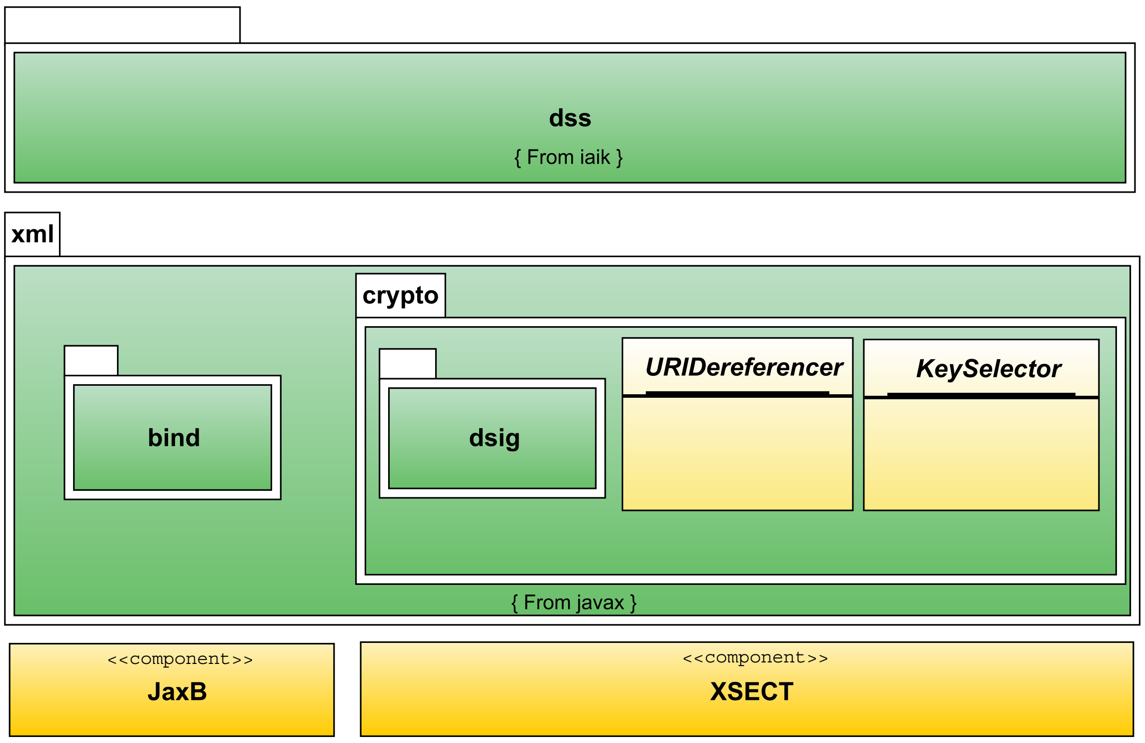


Figure 5.1.1: Architecture

This section describes the modular principle and structures of the *OASIS-DSS* software library we created for this thesis. We show that a *OASIS-DSS* library can be implemented in Java™ using redistributable software components. *OASIS-DSS* is a very open standard and provides a lot of freedom for implementations. The presented *OASIS-DSS* library implementation is built on top of Java Architecture for XML Binding (*JAXB*) and XML Security Toolkit (*XSECT*) as can be seen in Figure 5.1.1. *JAXB* and *XSECT* offer *APIs* mostly in the `javax.xml` package and *OASIS-DSS* makes extensive use of them.

XSECT in an *XMLDSIG* library that implements the Java Specification Request 105 XML Digital Signature APIs (*JSR105*) *API*. It is currently maintained by the author of this thesis.

JAXB is a standard *API* defined by the Java™ community process 222 and generates Java™ code consisting of classes representing the *Schema* and a run-time and for marshalling and unmarshalling. *JAXB* contains a *Schema* compiler that generates the set of classes reflecting the *Schema* in a class hierarchy. It further generates a run-time that parses the *XML* and via *SAX* creates object instances reflecting the data.

A very important part is the `KeySelector`, because an *OASIS-DSS* library cannot make any general choice about key storage and protection. The `KeySelector` hence is the interface for application developers to plug in their key management and certificate revocation checking. *XSECT* is scheduled to contain complete certificate path validation within the first quarter of 2009 when the *IAIK PKI* module will become available.

The `URIDereferencer` plays an equally outstanding role as *OASIS-DSS* dereferences *URIs* within its protocol structure <dss:InputDocuments>.

5.2 Design

A hierarchical processor model has been chosen that coarsely reflects the structure of *OASIS-DSS*. *JAXB* generates the binding classes from the *OASIS-DSS Schema*. A binding file (explained later) is used to associate implementation classes, which are extended by the classes generated with an *XML* binding framework. The classes have been generated on the level of the *Schema* anonymous types where possible or particles otherwise. These implementation classes written by the author of this thesis are polymorphic by their "`getProcessedBy(DssXpProcessor proc)`" method and "know" what processor will handle them. By means of the binding file these have been injected into the type hierarchy of the "entity" classes reflecting the *OASIS-DSS Schema*.
The processors in turn always have a "`process(DssXpProcessable obj)`" method, thus avoiding to walk the object tree. It is more like the tree of unmarshalled objects walks down the processors and the processors extract the needed information to generate the signature or verify it.

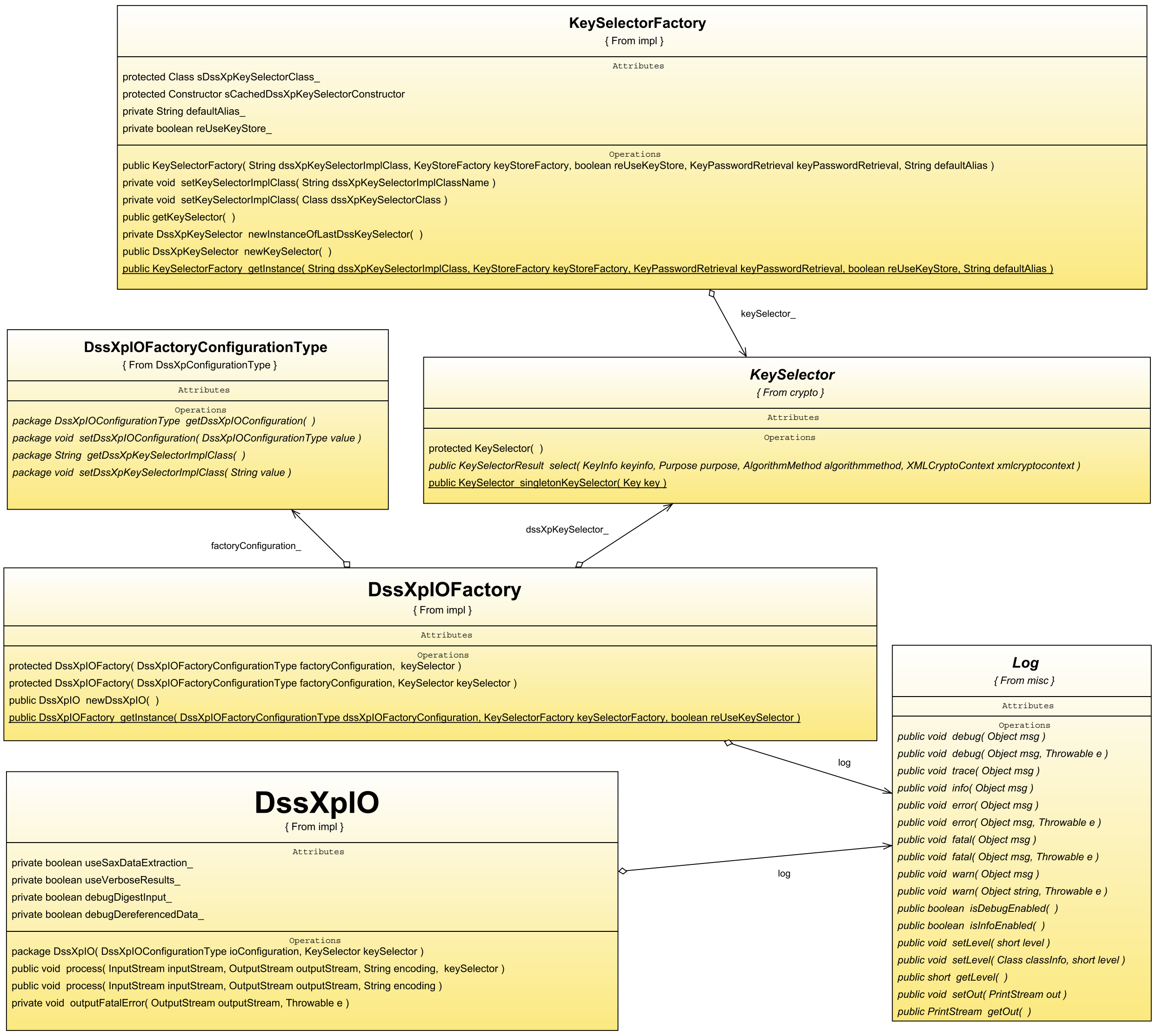
5.2.1 Components

`DssXpIOFactory` in Figure 5.2.1 produces `DssXpIO` Objects and each of them handles one request, then responds and then dies. It receives a configuration and about what instance class implementing `KeySelector` shall be instantiated by the `KeySelectorFactory`.

`DssXpIOFactoryConfigurationType` - is the top level configuration of the *OASIS-DSS* library.

`DssXpIO` - Dss *XML* processor Input Output. It is the core element an handles one request pulled from an `InputStream` and writes to an `OutputStream`.

`KeySelectorFactory` - has a default implementation, which may however be overridden by means of a configuration file. It provides the *OASIS-DSS* library with a *JSR105* `KeySelector` when needed.



KeySelectorFactory
{ From impl }
Attributes
protected Class sDssXpKeySelectorClass_
protected Constructor sCachedDssXpKeySelectorConstructor
private String defaultAlias_
private boolean reUseKeyStore_
Operations
public KeySelectorFactory(String dssXpKeySelectorImplClass, KeyStoreFactory keyStoreFactory, boolean reUseKeyStore, KeyPasswordRetrieval keyPasswordRetrieval, String defaultAlias)
private void setKeySelectorImplClass(String dssXpKeySelectorImplClassName)
private void setKeySelectorImplClass(Class dssXpKeySelectorClass)
public getKeySelector( )
private DssXpKeySelector newInstanceOfLastDssKeySelector( )
public DssXpKeySelector newKeySelector( )
public KeySelectorFactory getInstance(String dssXpKeySelectorImplClass, KeyStoreFactory keyStoreFactory, KeyPasswordRetrieval keyPasswordRetrieval, boolean reUseKeyStore, String defaultAlias)
keySelector_
DssXpIOFactoryConfigurationType
{ From DssXpConfigurationType }
Attributes
Operations
package DssXpIOConfigurationType getDssXpIOConfiguration( )
package void setDssXpIOConfiguration(DssXpIOConfigurationType value)
package String getDssXpKeySelectorImplClass( )
package void setDssXpKeySelectorImplClass(String value)
KeySelector
{ From crypto }
Attributes
Operations
protected KeySelector( )
public KeySelectorResult select(KeyInfo keyinfo, Purpose purpose, AlgorithmMethod algorithmmethod, XMLCryptoContext xmlcryptocontext)
public KeySelector singletonKeySelector(Key key)
factoryConfiguration_
dssXpKeySelector_
DssXpIOFactory
{ From impl }
Attributes
Operations
protected DssXpIOFactory(DssXpIOFactoryConfigurationType factoryConfiguration, keySelector)
protected DssXpIOFactory(DssXpIOFactoryConfigurationType factoryConfiguration, KeySelector keySelector)
public DssXpIO newDssXpIO( )
public DssXpIOFactory getInstance(DssXpIOFactoryConfigurationType dssXpIOFactoryConfiguration, KeySelectorFactory keySelectorFactory, boolean reUseKeySelector)
log
Log
{ From misc }
Attributes
Operations
public void debug(Object msg)
public void debug(Object msg, Throwable e)
public void trace(Object msg)
public void info(Object msg)
public void error(Object msg)
public void error(Object msg, Throwable e)
public void fatal(Object msg)
public void fatal(Object msg, Throwable e)
public void warn(Object msg)
public void warn(Object string, Throwable e)
public boolean isDebugEnabled( )
public boolean isInfoEnabled( )
public void setLevel(short level)
public void setLevel(Class classInfo, short level)
public short getLevel( )
public void setOut(PrintStream out)
public PrintStream getOut( )
DssXpIO
{ From impl }
Attributes
private boolean useSaxDataExtraction_
private boolean useVerboseResults_
private boolean debugDigestInput_
private boolean debugDereferencedData_
Operations
package DssXpIO(DssXpIOConfigurationType ioConfiguration, KeySelector keySelector)
public void process(InputStream inputStream, OutputStream outputStream, String encoding, keySelector)
public void process(InputStream inputStream, OutputStream outputStream, String encoding)
private void outputFatalError(OutputStream outputStream, Throwable e)
log

Figure 5.2.1: The DssXpIOFactory produces DssXpIO Objects.

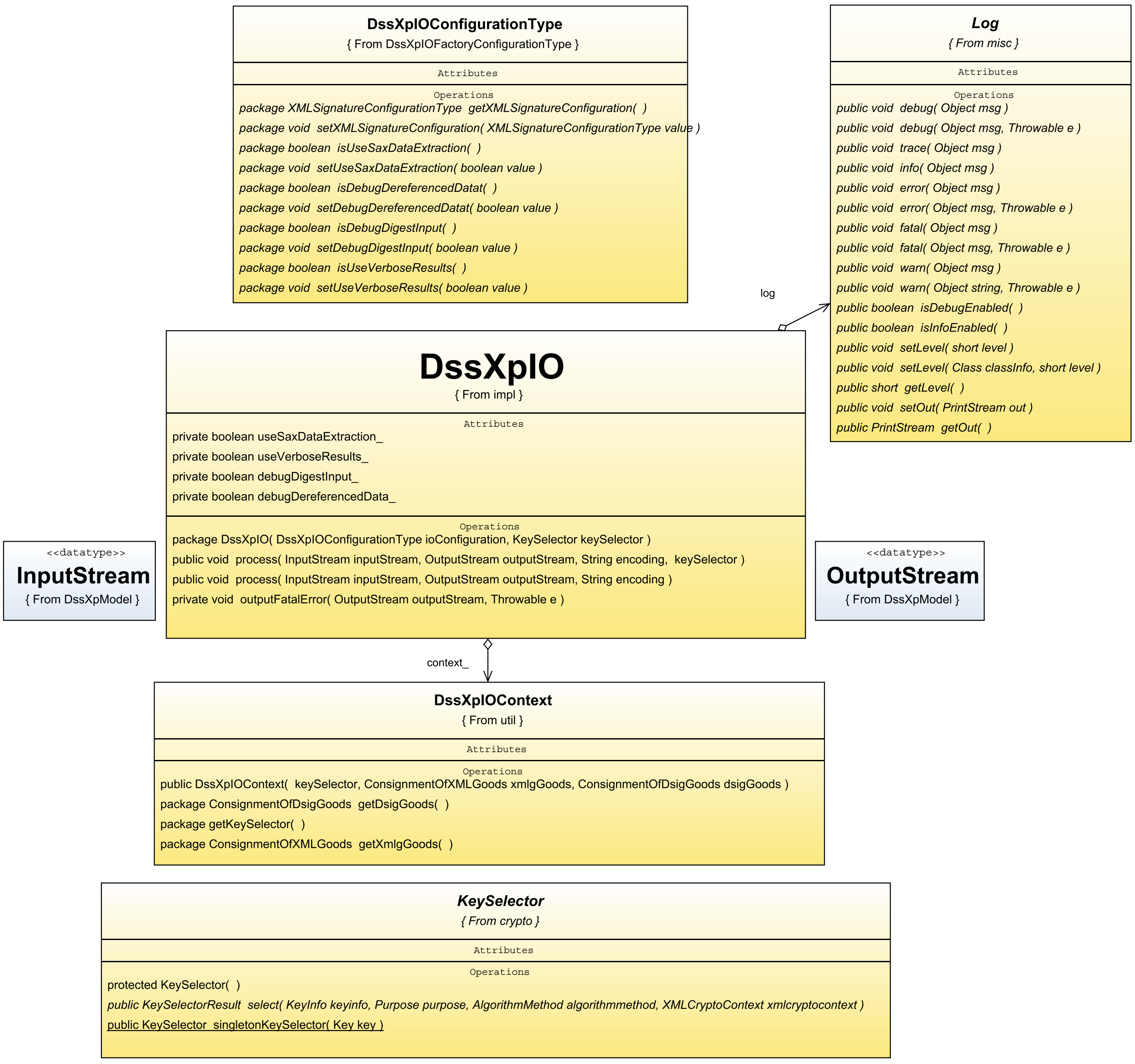

DssXpIOConfigurationType
{ From DssXpIOFactoryConfigurationType }
Attributes
Operations
package XMLSignatureConfigurationType getXMLSignatureConfiguration( )
package void setXMLSignatureConfiguration(XMLSignatureConfigurationType value)
package boolean isUseSaxDataExtraction( )
package void setUseSaxDataExtraction(boolean value)
package boolean isDebugDereferencedDatat( )
package void setDebugDereferencedDatat(boolean value)
package boolean isDebugDigestInput( )
package void setDebugDigestInput(boolean value)
package boolean isUseVerboseResults( )
package void setUseVerboseResults(boolean value)
Log
{ From misc }
Attributes
Operations
public void debug(Object msg)
public void debug(Object msg, Throwable e)
public void trace(Object msg)
public void info(Object msg)
public void error(Object msg)
public void error(Object msg, Throwable e)
public void fatal(Object msg)
public void fatal(Object msg, Throwable e)
public void warn(Object msg)
public void warn(Object string, Throwable e)
public boolean isDebugEnabled( )
public boolean isInfoEnabled( )
public void setLevel(short level)
public void setLevel(Class classInfo, short level)
public short getLevel( )
public void setOut(PrintStream out)
public PrintStream getOut( )
log
DssXpIO
{ From impl }
Attributes
private boolean useSaxDataExtraction_
private boolean useVerboseResults_
private boolean debugDigestInput_
private boolean debugDereferencedData_
Operations
package DssXpIO(DssXpIOConfigurationType ioConfiguration, KeySelector keySelector)
public void process(InputStream inputStream, OutputStream outputStream, String encoding, keySelector)
public void process(InputStream inputStream, OutputStream outputStream, String encoding)
private void outputFatalError(OutputStream outputStream, Throwable e)
<<datatype>>
InputStream
{ From DssXpModel }
<<datatype>>
OutputStream
{ From DssXpModel }
context_
DssXpIOContext
{ From util }
Attributes
Operations
public DssXpIOContext(keySelector, ConsignmentOfXMLGoods xmlgGoods, ConsignmentOfDsigGoods dsigGoods)
package ConsignmentOfDsigGoods getDsigGoods( )
package getKeySelector( )
package ConsignmentOfXMLGoods getXmlgGoods( )
KeySelector
{ From crypto }
Attributes
Operations
protected KeySelector( )
public KeySelectorResult select(KeyInfo keyinfo, Purpose purpose, AlgorithmMethod algorithmmethod, XMLCryptoContext xmlcryptocontext)
public KeySelector singletonKeySelector(Key key)

Figure 5.2.2: A DssXpIO Objects, processes an InputStream and writes to an Outputstream.

DssXpIO

A `DssXpIO` processes one request, responds and then dies.

`DssXpIOConfigurationType` - is the request level configuration of the *OASIS-DSS* library. The default signature configuration and whether payload shall be extracted at *SAX* level can be configured in this object. Optional outputs providing debug information about the `DereferencedData` and `DigestInput` can be enabled.

`DssXpIOContext` - holds the `KeySelector` and refers to *JSR105* and *JAXP* libraries, capsuled under the terms `ConsignmentOfDsigGoods` and `ConsignmentOfXMLGoods` respectively.

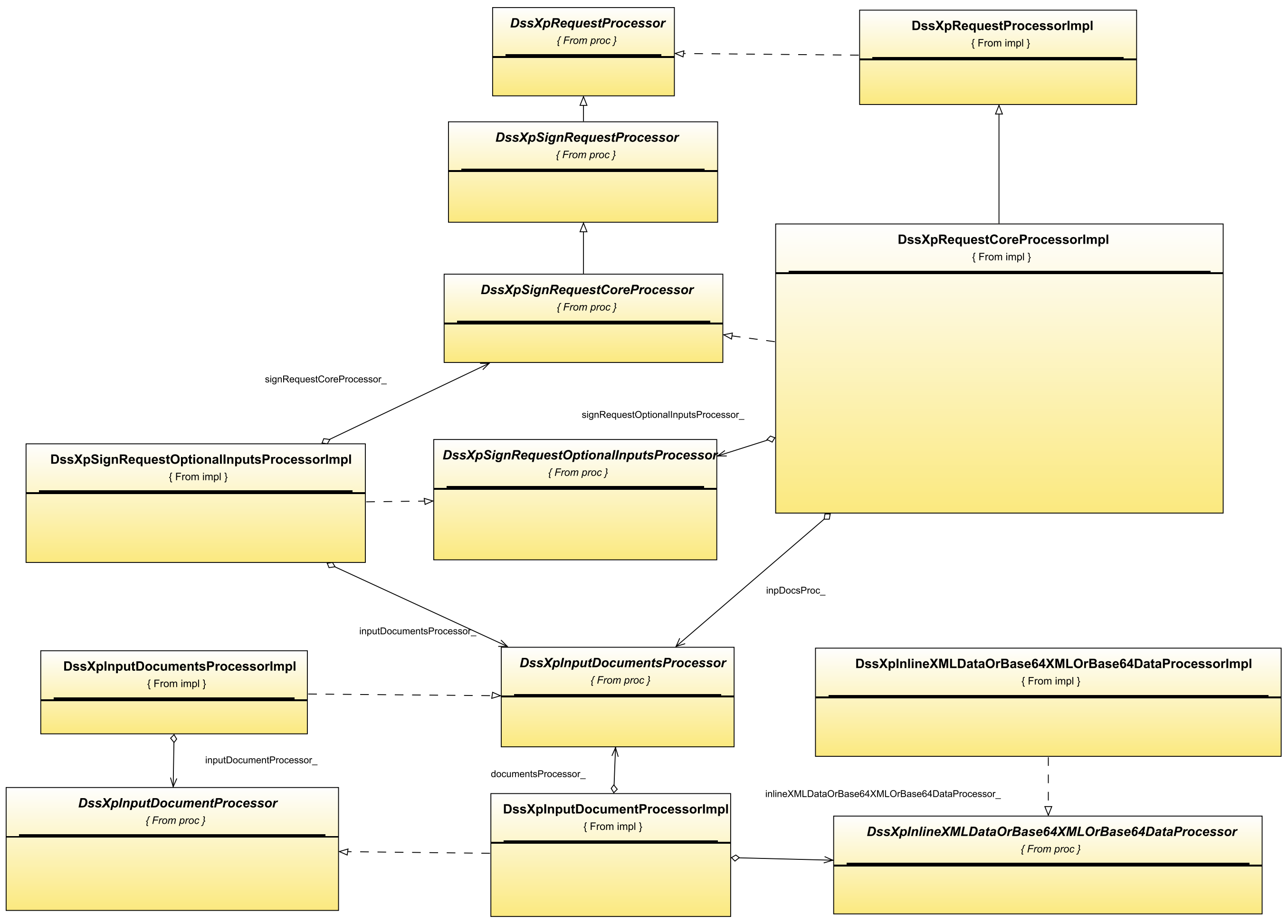

DssXpRequestProcessor
{ From proc }
DssXpRequestProcessorImpl
{ From impl }
DssXpSignRequestProcessor
{ From proc }
DssXpRequestCoreProcessorImpl
{ From impl }
DssXpSignRequestCoreProcessor
{ From proc }
signRequestCoreProcessor_
signRequestOptionalInputsProcessor_
DssXpSignRequestOptionalInputsProcessorImpl
{ From impl }
DssXpSignRequestOptionalInputsProcessor
{ From proc }
inpDocsProc_
inputDocumentsProcessor_
DssXpInputDocumentsProcessorImpl
{ From impl }
DssXpInputDocumentsProcessor
{ From proc }
DssXpInlineXMLDataOrBase64XMLOrBase64DataProcessorImpl
{ From impl }
inputDocumentProcessor_
documentsProcessor_
inlineXMLDataOrBase64XMLOrBase64DataProcessor_
DssXpInputDocumentProcessor
{ From proc }
DssXpInputDocumentProcessorImpl
{ From impl }
DssXpInlineXMLDataOrBase64XMLOrBase64DataProcessor
{ From proc }

Figure 5.2.3: Processors

5.2.2 Processors

`DssXpRequesCoreProcessor(Impl)` - gets passed control from `DssXpIO`. It is however not held by any variable in `DssXpIO` so it can be garbage collected as soon as it finished its work to process the request. It has an `DssXpSignRequestOptionalInputsProcessor` which first evaluates the optional inputs to configure other processors.

The core processor further has a `DssXpInputDocumentsProcessor`,which performs the basic processing as specified in Figure 3.3.1 on page 60, yet allowing several hooks and methods to amend basic processing.

`DssXpSignRequestOptionalInputsProcessor` - has backward references to the `DssXpRequestCoreProcessor` and to the `DssXpInputDocumentsProcessor` to be able to amend the basic processing as indicated by optional inputs.

`DssXpInputDocumentsProcessor` - is the workhorse of this implementation with respect to the actual *XMLDSIG* signature creation.

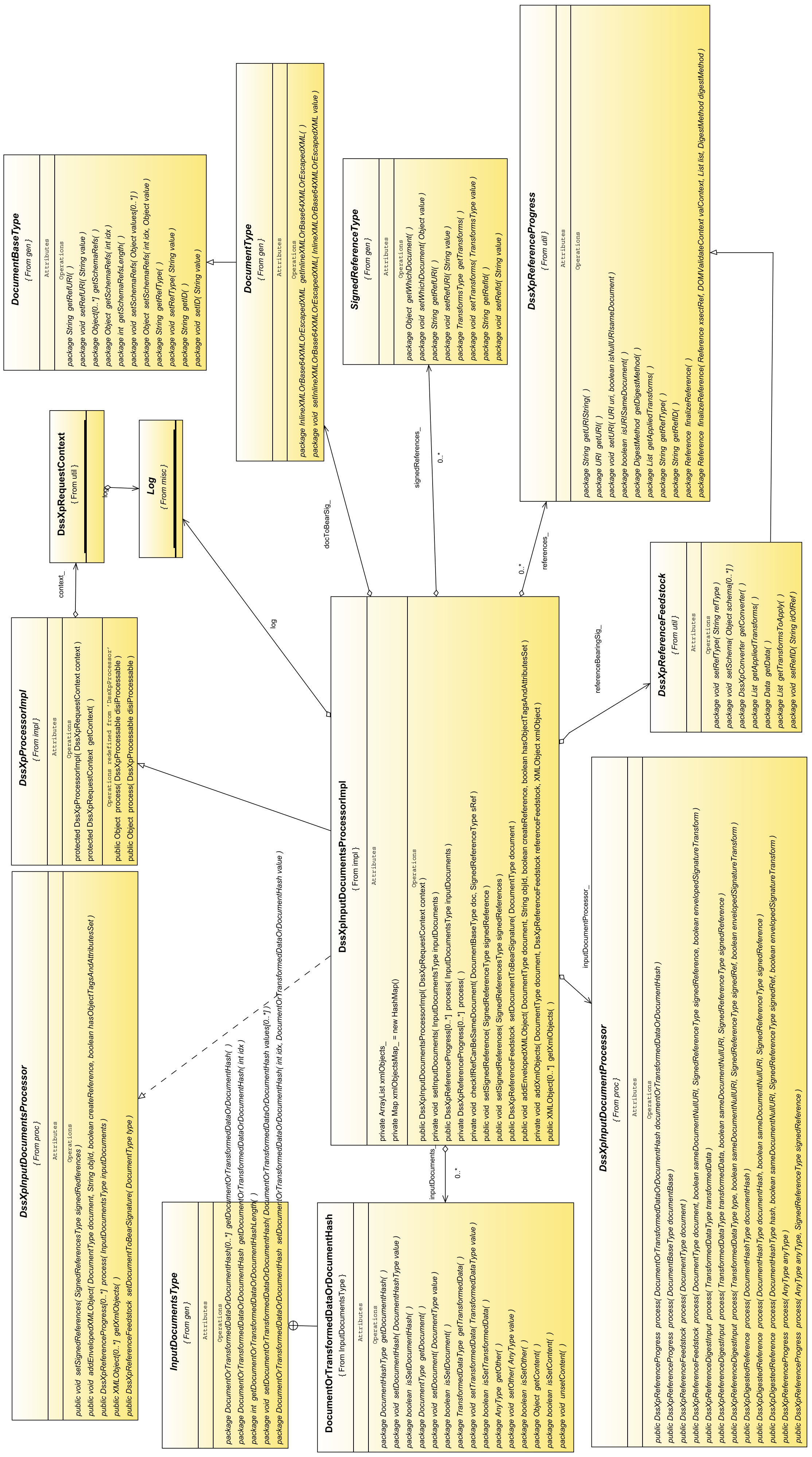

DocumentBaseType { From gen }
Attributes
Operations
package String getRefURI( )
package void setRefURI(String value)
package Object[0..*] getSchemaRefs( )
package Object getSchemaRefs(int idx)
package int getSchemaRefsLength( )
package void setSchemaRefs(Object values[0..*])
package Object setSchemaRefs(int idx, Object value)
package String getRefType( )
package void setRefType(String value)
package String getID( )
package void setID(String value)
DocumentType { From gen }
Attributes
Operations
package InlineXMLOrBase64XMLOrEscapedXML getInlineXMLOrBase64XMLOrEscapedXML( )
package void setInlineXMLOrBase64XMLOrEscapedXML(InlineXMLOrBase64XMLOrEscapedXML value)
SignedReferenceType { From gen }
Attributes
Operations
package Object getWhichDocument( )
package void setWhichDocument(Object value)
package String getRefURI( )
package void setRefURI(String value)
package TransformsType getTransforms( )
package void setTransforms(TransformsType value)
package String getRefId( )
package void setRefId(String value)
DssXpReferenceProgress { From util }
Attributes
Operations
package String getURIString( )
package URI getURI( )
package void setURI(URI uri, boolean isNullURIsameDocument)
package boolean isURISameDocument( )
package DigestMethod getDigestMethod( )
package List getAppliedTransforms( )
package String getRefType( )
package String getRefID( )
package Reference finalizeReference( )
package Reference finalizeReference(Reference xsectRef, DOMValidateContext valContext, List list, DigestMethod digestMethod)
DssXpRequestContext { From util }
log
Log { From misc }
context_
log
docToBearSig_
signedReferences_
0..*
references_
0..*
DssXpProcessorImpl { From impl }
Attributes
Operations
protected DssXpProcessorImpl(DssXpRequestContext context)
protected DssXpRequestContext getContext( )
Operations redefined from 'DssXpProcessor'
public Object process(DssXpProcessable dsiProcessable)
public Object process(DssXpProcessable dsiProcessable)
DssXpInputDocumentsProcessorImpl { From impl }
Attributes
private ArrayList xmlObjects_
private Map xmlObjectsMap_ = new HashMap()
Operations
public DssXpInputDocumentsProcessorImpl(DssXpRequestContext context)
private void setInputDocuments(InputDocumentsType inputDocuments)
public DssXpReferenceProgress[0..*] process(InputDocumentsType inputDocuments)
private DssXpReferenceProgress[0..*] process( )
private void checkIfRefCanBeSameDocument(DocumentBaseType doc, SignedReferenceType sRef)
public void setSignedReference(SignedReferenceType signedReference)
public void setSignedReferences(SignedReferencesType signedReferences)
public DssXpReferenceFeedstock setDocumentToBearSignature(DocumentType document)
public void addEnvelopedXMLObject(DocumentType document, String objId, boolean createReference, boolean hasObjectTagsAndAttributesSet)
private void addXmlObjects(DocumentType document, DssXpReferenceFeedstock referenceFeedstock, XMLObject xmlObject)
public XMLObject[0..*] getXmlObjects( )
referenceBearingSig_
DssXpReferenceFeedstock { From util }
Attributes
Operations
package void setRefType(String refType)
package void setSchema(Object schema[0..*])
package DssXpConverter getConverter( )
package List getAppliedTransforms( )
package Data getData( )
package List getTransformsToApply( )
package void setRefID(String idOfRef)
inputDocumentProcessor_
DssXpInputDocumentsProcessor { From proc }
Attributes
Operations
public void setSignedReferences(SignedReferencesType signedReferences)
public void addEnvelopedXMLObject(DocumentType document, String objId, boolean createReference, boolean hasObjectTagsAndAttributesSet)
public DssXpReferenceProgress[0..*] process(InputDocumentsType inputDocuments)
public XMLObject[0..*] getXmlObjects( )
public DssXpReferenceFeedstock setDocumentToBearSignature(DocumentType type)
InputDocumentsType { From gen }
Attributes
Operations
package DocumentOrTransformedDataOrDocumentHash[0..*] getDocumentOrTransformedDataOrDocumentHash( )
package DocumentOrTransformedDataOrDocumentHash getDocumentOrTransformedDataOrDocumentHash(int idx)
package int getDocumentOrTransformedDataOrDocumentHashLength( )
package void setDocumentOrTransformedDataOrDocumentHash(DocumentOrTransformedDataOrDocumentHash values[0..*])
package DocumentOrTransformedDataOrDocumentHash setDocumentOrTransformedDataOrDocumentHash(int idx, DocumentOrTransformedDataOrDocumentHash value)
inputDocuments_
0..*
DocumentOrTransformedDataOrDocumentHash { From InputDocumentsType }
Attributes
Operations
package DocumentHashType getDocumentHash( )
package void setDocumentHash(DocumentHashType value)
package boolean isSetDocumentHash( )
package DocumentType getDocument( )
package void setDocument(DocumentType value)
package boolean isSetDocument( )
package TransformedDataType getTransformedData( )
package void setTransformedData(TransformedDataType value)
package boolean isSetTransformedData( )
package AnyType getOther( )
package void setOther(AnyType value)
package boolean isSetOther( )
package Object getContent( )
package boolean isSetContent( )
package void unsetContent( )
DssXpInputDocumentProcessor { From proc }
Attributes
Operations
public DssXpReferenceProgress process(DocumentOrTransformedDataOrDocumentHash documentOrTransformedDataOrDocumentHash)
public DssXpReferenceProgress process(DocumentBaseType documentBase)
public DssXpReferenceFeedstock process(DocumentType document)
public DssXpReferenceFeedstock process(DocumentType document, boolean sameDocumentNullURI, SignedReferenceType signedReference, boolean envelopedSignatureTransform)
public DssXpReferenceDigestInput process(TransformedDataType transformedData)
public DssXpReferenceDigestInput process(TransformedDataType transformedData, boolean sameDocumentNullURI, SignedReferenceType signedReference)
public DssXpReferenceDigestInput process(TransformedDataType type, boolean sameDocumentNullURI, SignedReferenceType signedRef, boolean envelopedSignatureTransform)
public DssXpDigestedReference process(DocumentHashType documentHash)
public DssXpDigestedReference process(DocumentHashType documentHash, boolean sameDocumentNullURI, SignedReferenceType signedReference)
public DssXpDigestedReference process(DocumentHashType hash, boolean sameDocumentNullURI, SignedReferenceType signedRef, boolean envelopedSignatureTransform)
public DssXpReferenceProgress process(AnyType anyType)
public DssXpReferenceProgress process(AnyType anyType, SignedReferenceType signedReference)

Figure 5.2.4: DssXpInputDocumentsProcessor

Fidelity modes - DssXpInlineOrBase64XMLOrBase64DataProcessor

`DssXpInlineOrBase64XMLOrBase64DataProcessor` from Figure 5.2.4 on page 107 is unpacking and packing the payload in the various fidelity modes ranging from <dss:Base64XML> over <dss:EscapedXML> to <dss:InlineXML>.

`DssXpInputDocumentProcessor` prepares one `DssXpReferenceProgress` per document.

Stages - DssXpReferenceProgress

`DssXpReferenceProgress` matches the various stages of the chain of <ds:Transforms> from Figure 3.3.2 on page 60 shown in Figure 5.2.5 on page 109 from left to right.

5.2.3 `URIDerferencer` Decoration

To dereference data objects in the protocol, the `URIDerferencer` of the given *XMLDSIG* implementation was decorated by the `DssXpUriDereferencer` and its subtypes. Depending on which mechanism (*JAXB*, Sax Extraction) is to be used the `DssXpUriDereferencer` dereferences either documents extracted on the *SAX* level or allows to use the extraction mechanism of *JAXB*.

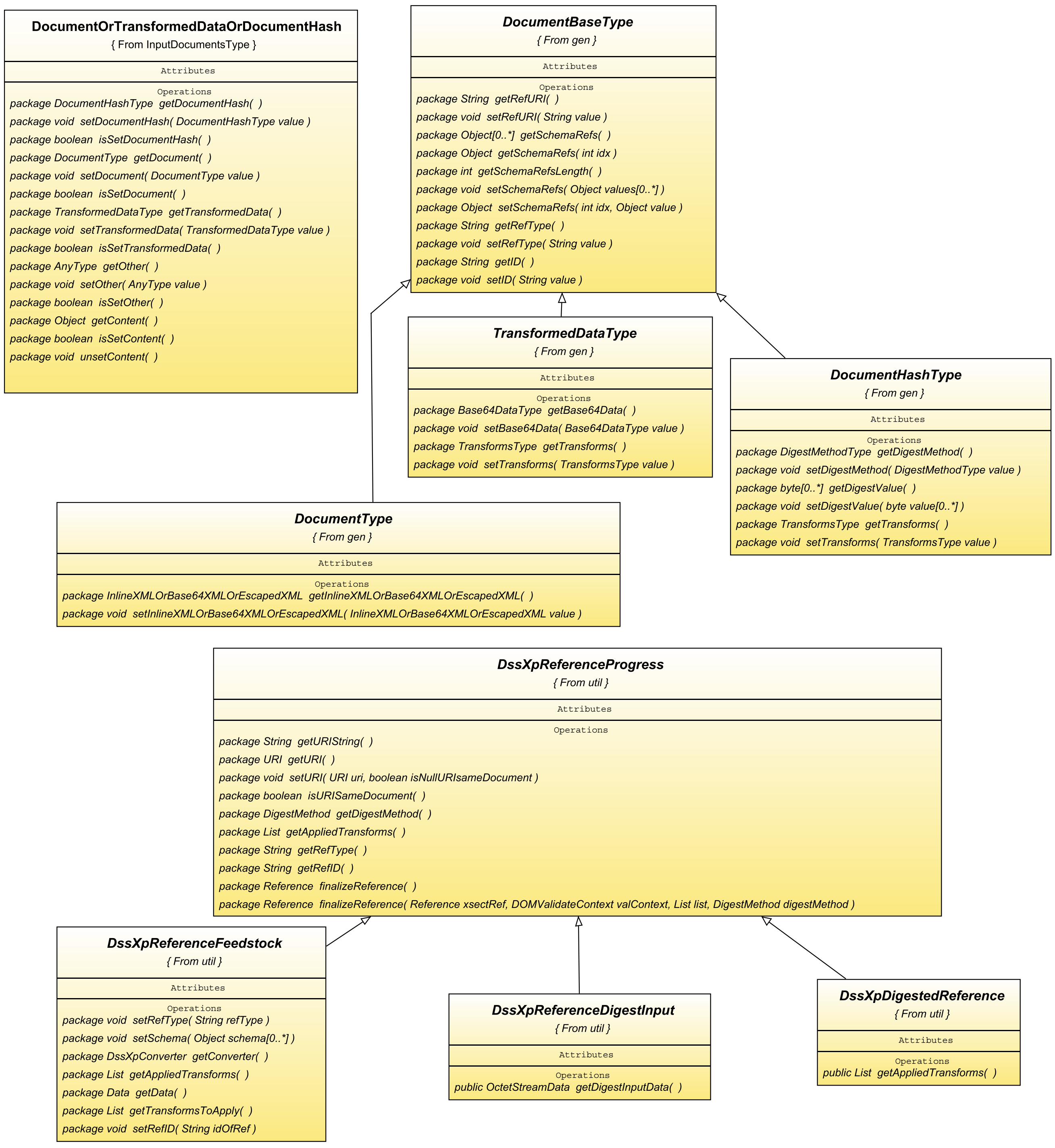

DocumentOrTransformedDataOrDocumentHash
{ From InputDocumentsType }
Attributes
Operations
package DocumentHashType getDocumentHash( )
package void setDocumentHash(DocumentHashType value)
package boolean isSetDocumentHash( )
package DocumentType getDocument( )
package void setDocument(DocumentType value)
package boolean isSetDocument( )
package TransformedDataType getTransformedData( )
package void setTransformedData(TransformedDataType value)
package boolean isSetTransformedData( )
package AnyType getOther( )
package void setOther(AnyType value)
package boolean isSetOther( )
package Object getContent( )
package boolean isSetContent( )
package void unsetContent( )
DocumentBaseType
{ From gen }
Attributes
Operations
package String getRefURI( )
package void setRefURI(String value)
package Object[0..*] getSchemaRefs( )
package Object getSchemaRefs(int idx)
package int getSchemaRefsLength( )
package void setSchemaRefs(Object values[0..*])
package Object setSchemaRefs(int idx, Object value)
package String getRefType( )
package void setRefType(String value)
package String getID( )
package void setID(String value)
TransformedDataType
{ From gen }
Attributes
Operations
package Base64DataType getBase64Data( )
package void setBase64Data(Base64DataType value)
package TransformsType getTransforms( )
package void setTransforms(TransformsType value)
DocumentHashType
{ From gen }
Attributes
Operations
package DigestMethodType getDigestMethod( )
package void setDigestMethod(DigestMethodType value)
package byte[0..*] getDigestValue( )
package void setDigestValue(byte value[0..*])
package TransformsType getTransforms( )
package void setTransforms(TransformsType value)
DocumentType
{ From gen }
Attributes
Operations
package InlineXMLOrBase64XMLOrEscapedXML getInlineXMLOrBase64XMLOrEscapedXML( )
package void setInlineXMLOrBase64XMLOrEscapedXML(InlineXMLOrBase64XMLOrEscapedXML value)
DssXpReferenceProgress
{ From util }
Attributes
Operations
package String getURIString( )
package URI getURI( )
package void setURI(URI uri, boolean isNullURIsameDocument)
package boolean isURISameDocument( )
package DigestMethod getDigestMethod( )
package List getAppliedTransforms( )
package String getRefType( )
package String getRefID( )
package Reference finalizeReference( )
package Reference finalizeReference(Reference xsectRef, DOMValidateContext valContext, List list, DigestMethod digestMethod)
DssXpReferenceFeedstock
{ From util }
Attributes
Operations
package void setRefType(String refType)
package void setSchema(Object schema[0..*])
package DssXpConverter getConverter( )
package List getAppliedTransforms( )
package Data getData( )
package List getTransformsToApply( )
package void setRefID(String idOfRef)
DssXpReferenceDigestInput
{ From util }
Attributes
Operations
public OctetStreamData getDigestInputData( )
DssXpDigestedReference
{ From util }
Attributes
Operations
public List getAppliedTransforms( )

Figure 5.2.5: ReferenceProgress

5.3 Data binding

An *OASIS-DSS* server needs to process the *XML* data it receives and an *OASIS-DSS* library can approach this task at different levels. Taking a look at the architecture again, we will see that *JSR105* is designed for *DOM* and *XSECT* is hence built on top of *DOM*. This becomes apparent because they support random access to all parts of the document, which usually involves having the complete *XML* structure in memory as in *DOM*.

Given the structure of an *OASIS-DSS* request we however note that the optional inputs at the beginning will always refer to a dss:DocumentBaseType further on in the *document order*. Especially in the case of a <dss:Document> without <ds:Transforms>, *C14n11*[1] plus a digest method could be applied directly on a *SAX* level. Similarly in the basic processing only a digest method would be applied directly for <dss:TransformedData>. This would not even require the server to have the document ever in memory. Recalling what we discussed however in section 3.3.2.1 one should do such only with "known good" values.

$$\begin{array}{lclcl}
\mathit{XML} \rightarrow \mathit{SAX} \rightarrow & & \rightarrow \text{Processor} \rightarrow & & \rightarrow \mathit{SAX} \rightarrow \mathit{XML} \\
\mathit{XML} \rightarrow \mathit{SAX} \rightarrow & \text{gen. code} & \rightarrow \text{Processor} \rightarrow & \text{gen. code} & \rightarrow \mathit{SAX} \rightarrow \mathit{XML} \\
\mathit{XML} \rightarrow \mathit{SAX} \rightarrow & \mathit{DOM} & \rightarrow \text{Processor} \rightarrow & \mathit{DOM} & \rightarrow \mathit{SAX} \rightarrow \mathit{XML}
\end{array}$$

Figure 5.3.1: Unmarshalling → Processing → Marshalling

We have to recognize that at the time we started developing this library neither *SAX* or *StAX* based *XMLDSIG* nor such *C14n11* implementations were available as of end 2008. The development of such was considered out of scope for this thesis and constitutes potential future work.

A pure *SAX* based processing, albeit feasible, if only the basic processing has to be implemented, would have the disadvantage of only being able to offer event based interfaces to higher level server logic. Thus for flexibility and extensibility it is not the best choice for an implementation.

XSECT for instance uses a registry of classes matching the schema definitions of the Garden of Eden Style (subsection 2.2.2.3) grammar of *XMLDSIG*. These custom classes are not very strictly typed and constitute a considerable amount of code to be maintained. They are operating on an underlying *DOM* tree.

An alternative to the "hard coded" *DOM* marshalling and unmarshalling was hence looked into and the following binding frameworks were available at the time the implementation begun.

Castor, Javolution, *JAXB*, JBind, XMLBeans [19] are partly providing full data typing. As *OASIS-DSS* at the time the implementation begun was still in the flux, a too tight coupling between library code and

[1]We recall that *C14n* has been superseded by *C14n11* (subsection 2.5.2), and in the case on <dss:InlineXML> *Exc-C14n* shall be used.

standard grammar [19] was not desired. A data binding framework promises to require less memory than *DOM* and faster access to data.

The use of an *XML* binding framework was considered as a *Schema* is available for *OASIS-DSS*.

An "*XML* binding framework" is a concept to represent *XML* in memory and perform the tasks of marshalling and unmarshalling. It usually offers typing and typed access to attributes and tree structure. Interestingly the *document order* is not preserved in a framework like *JAXB*. Which in the case of *OASIS-DSS* is uncritical - with the exception of the payload.

> Marshalling is the process of generating an XML representation for an object in memory. As with Java object serialization, the representation needs to include all dependent objects: objects referenced by our main object, objects referenced by those objects, and so on.[19]

> Unmarshalling is the reverse process of marshaling, building an object (and potentially a graph of linked objects) in memory from an XML representation. [19]

For the payload a concept called "*DOM* binding" is offered by *JAXB* and was used to bind the actual payload in *OASIS-DSS* as parsed *DOM* trees.
A hierarchical structured processor approach, only coarsely reflecting the structure of *OASIS-DSS*, was preferred against an implementation putting logic directly in the entity classes. Code re-generation from a changed *Schema* should only affect methods in charge for processing the relevant elements or types.
JAXB was recommended in [19] and an important factor was that *JAXB's* redistributable components should be licensable for royalty free commercial use. This is the case as a reference implementation of the *JAXB API* is available in the Java™ Web Service Development Pack (JWSDP) 1.6[2].

JAXB's vendor customizations also promised to be able to subset the elements to be generated and types as *OASIS-DSS* amongst two foreign *Schemas* imports the complete *XMLDSIG Schema*, which is covered by *JSR105* implementations already. It supports "*DOM* binding" for payload extraction as well. *JAXB* also in the long run has the advantage of being a standard *API*. Hence an implementation could potentially be ported to another implementation, which is because of extensive need for vendor customization not certain.

[2]"`http://java.sun.com/webservices/docs/1.6/jaxb/`"
JAXB 2.0 was not available at the time the implementation started and introduces a dependency to Java™1.5 or higher.

5.3.1 Java API for XML Binding (*JAXB*)

JAXB supports to bind implantation classes, which has been used extensively.
Figure 5.3.2 shows how an element bound to an implementation class.

```
<jxb:bindings node="/xs:schema/xs:complexType[@name='InlineXMLType']">
  <jxb:class name="InlineXMLType" implClass="iaik.dss.procable.
      DssXpInlineXMLTypeImpl"/>
</jxb:bindings>
```

Figure 5.3.2: Binding an Implementation class

JAXB supports two kinds of customizations [82]:

- In-line *Schema* Customizations using `xs:annotation/xs:appinfo`
- External *Schema* Customizations using a Binding Customization Files

As it seems cleaner to have the *Schema* separated from the binding the second approach was taken.
The *DOM* vendor customization can be used to access wild-card content or other parts, which are either annotated or bound using an external bindings file. *JAXB* generates a *DOM* binding by default for accessing wild-card content with ”lax” handling and unknown content is discarded, the `<xjc:dom/>` customization however preserves all content. [83]

DOM - Binding

Figure 5.3.3 shows how a wild-card is mapped by a *DOM* binding that has been annotated directly in the *Schema* to bind a wild-card to a *DOM* document.

```
...
   <xs:complexType>
      <xs:sequence>
         <xs:any>
            <xs:annotation>
              <xs:appinfo>
                <xjc:dom/>
              </xs:appinfo>
            </xs:annotation>
         </xs:any>
      </xs:sequence>
   </xs:complexType>
...
```

Figure 5.3.3: DOM Binding

JAXB's vendor customizations allow using a special element either directly in the *Schema* or in a bindings file. It tells *JAXB* that a certain part of the *Schema* shall not be unmarshalled, but rather be parsed into an in memory representation of an *XML* document. The customization element is called `xjc:dom` and can map parts of a *Schema* to a *DOM* tree representation. Wild-cards (`xs:any`), element declarations (`xs:element`), `xs:choice`, `xs:sequence` and other particles can be bound and represented as a *DOM* tree.

We identified in subsection 4.3.1 on page 96 the following <dss:InlineXML>, <ds:Transforms>, <ds:KeyInfo>, <ds:DigestMethod>, <ds:DigestValue> and <dss:SignatureObject>'s child structure <ds:Signature> to be handled as payload.
Extension points and optional inputs are also wild-cards, but are attempted to be parsed by *JAXB*.
Figure 5.3.4 shows how an element is mapped by a *DOM* binding in an external bindings file using an *XPath* expression to point to the corresponding location in the *Schema*.

```
<jxb:bindings node="/xs:schema/xs:element[@name='Signature']">
  <xjc:dom/>
</jxb:bindings>
```

Figure 5.3.4: DOM binding for <ds:Signature>

JAXB's DOM binding passes the complete namespace context down into the *DOM* tree model, which breaks all signatures not using *Exc-C14n*. At the time (WD30 page 15) this was a problem for all payload conveyed in *OASIS-DSS*. However there exists an option that allows not having to regenerate the run-time classes when the *Schema* classes are regenerated. Hence we were able to patch the run-time after intensive research and debugging of the code generated by *JAXB*. A class called `UnmarshallingEventHandlerAdaptor` was identified as causing the passing on of namespace context to the payload and has been patched.

JAXB's documentation [83] claims as mentioned above, that the `<xjc:dom/>` customization preserves all content. This is however not true for *processing instructions* and *comments*. At the time the implementation started this was not known and discovered by tests. Before that standardization effort, to include dss:InlineXMLType's attributes `ignorePIs` and `ignoreComments` were performed, a patch for *JAXB* was successfully researched.

Round-tripping

Albeit *XMLDSIG* signatures can be made robust against removed *comments* or *processing instructions* a service like *OASIS-DSS* has a responsibility not to change the data it received from the client. When applying an enveloped signature this should not imply that *comments* or *processing instructions* are removed from the document where the signature will be placed. One has to clearly distinguish between what is in the document and what the `DereferencedData` and `DigestInput` are.

In case of a <ds:Reference> dereferencing a *same-document reference* a node-set is retrieved ranging form containing all nodes including the signature and/or the signature's <ds:Object> to arbitrarily fragmented node-sets, everything is possible. If this node-set contains the nodes belonging to the signature at least those representing the <ds:DigestValue>s and the <ds:SignatureValue> have to be filtered or the signature cannot verify. So what is signed in such a case is a canonicalization of a document-subset. The document contains all nodes including such nodes that are not signed and have to be sent back to the client as such. Just because nodes are not signed does not mean they can be removed.

5.3.2 Round-tripping and *Infoset*

The enveloped and enveloping digital signature creation requires to preserve equivalence, as one has to work in place on the original document. Pure detached signature creation does not require to preserve equivalence as one does not work in-place and documents will not be returned by an *OASIS-DSS* library or server.
An author or user of a tool that generated some *XML* document that contains for instance an *XSLT* processing instruction in the *prolog* to display it, sends the document to a *OASIS-DSS* server to get it signed. This user would quite obviously not be very pleased, if adding the document to an enveloping signature or adding an enveloped *XMLDSIG* signature to the document, would strip off all of the *processing instructions* and *comments*.

A definition for round-tripping at the *SAX*level:

> Round-tripping a *SAX* event stream means turning it back into *XML* text and parsing the result, without losing any data. [23]

Using *base64 encoding* should be the first choice in such a situation, but in WD30 page 15 this was not the default and the case of transporting *XML* had to be solved.
Round tripping on a *SAX* level is already cumbersome[3], one has to register all Handlers (cf. 2.3.1) with the `XMLReader`. The `XMLReader` chunks well formed *XML* and "fires" the chunks at the event handlers. It is so to say the "XMLTokenizer".

Our Library defines an interface called `SAXRoundTripHandler`, that extends `ContentHandler`, `LexicalHandler`, `DeclHandler` and `DTDHandler` as we recall from subsection 2.3.1 that the information in an *XML* document is split up on the *SAX* level[4].

As *JAXB* is built on top of *SAX* the next layer above is *JAXB* itself. *JAXB* does mention round-tripping as a non-goal and with respect to round-tripping *Infosets* seem not to be a good choice any more for a transport protocol such as *OASIS-DSS*.

[3] A *SAX*parser allows to recreate the internal subset, but not any external parameter entities as they will be expanded. Conditional sections in external parsed entities are evaluated and declarations built up from parameter entities will be inlined[23]. Round tripping unexpanded entities, conditional sections in declarations which appear at the *SAX* level are assumed to be out of scope for this thesis. Especially as these cannot be navigated in the *XPath* data model.

[4]For <dss:InlineXML> the latter two could have been spared, but anticipating a technology that would allow to transport the *prolog* inside *XML* we extended those as well.

> Preserving equivalence of XML document when round tripping from XML document to Java representation and back to XML document again. While the *JAXB* specification does not require the preservation of the XML information set, it does not forbid the preservation of it. [78]

With a posting[5] to the *JAXB* users list we raised awareness for this problem when it comes to using *XMLDSIG* in binding frameworks. *JAXB* version 2.1 [79] moved, despite not putting a hard requirement, the following statements to its goals:

> 12. Preserving equivalence - Round tripping (Java to XML to Java) Transforming a Java content tree to XML content and back to Java content again should result in an equivalent Java content tree before and after the transformation.
>
> 13. Preserving equivalence - Round tripping (XML to Java to XML) While *JAXB* 1.0 specification did not require the preservation of the XML information set when round tripping from XML document to Java representation and back to XML document again, it did not forbid the preservation either. The same applies to this version of the specification. [79]

Round-tripping in *JAXB*

After intensive research the `W3CDOMUnmarshallingEventHandler` class was identified as using a `SAX2DOMEx` class that ignored *processing instructions* and *comments* and did not get all the *SAX* events passed.
Hence the *JAXB* run-time has been patched and the following classes are supplied by our library to assure, with maximum independence of the Java™ version or *XML* library version (*Xerces*/*Xalan*), that a maximum of exposure is achieved. Maximum exposure means in this context to get as many events as possible through to the *DOM* binding while underneath reading form a *SAX* stream.

`SAX2DOMEx` could be replaced by our libraries with a `SAX2DOMMaxDefaultHandler` which implements the previously mentioned `SAXRoundTripHandler` interface. It has a member called `_SAX2DOM` of type `Object` which will be one of the following depending on what is found on the Java™ class-path:

- `org.apache.xalan.xsltc.trax.SAX2DOM`
- `com.sun.org.apache.xalan.internal.xsltc.trax.SAX2DOM`
- `org.apache.xml.utils.DOMBuilder`
- `com.sun.org.apache.xml.internal.utils.DOMBuilder`
- `org.apache.xml.internal.utils.DOMBuilder`

[5] "`http://markmail.org/message/3kk5rq4ncq7frwfs`"

`_SAX2DOM` will be casted to whatever *SAX* handler is required and every *SAX* event is fired at it irrespective if it can handle it.
This allows it to be used despite not implementing the full functionality of `SAXRoundTripHandler` as long as no input appears that it cannot handle. If such input however causes it to fail the *OASIS-DSS* library is written in away where it catches all exceptions and exits gracefully with a response including the warning message from the exception and a new request may be supplied.

This allows a very stable use, despite such a flexible technique.

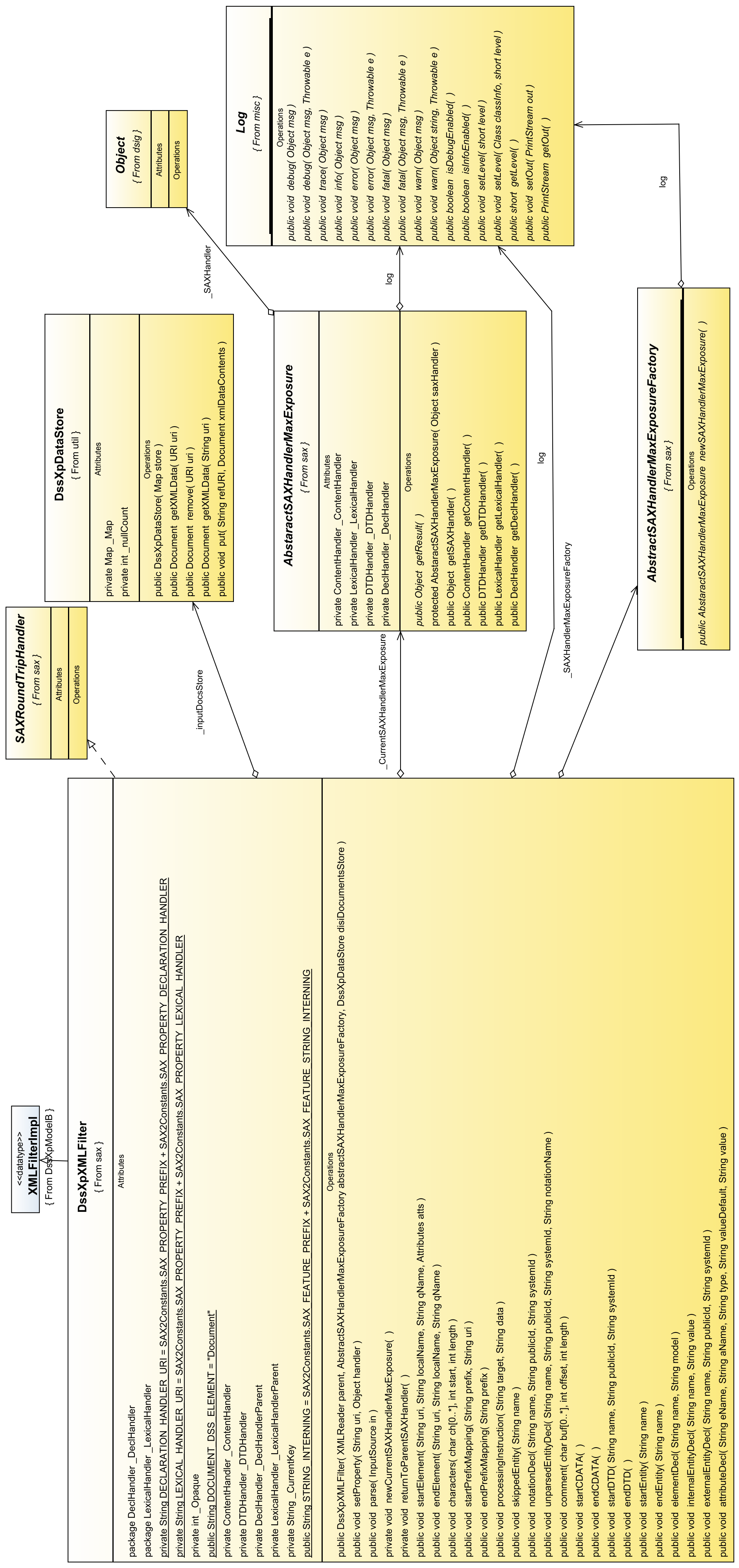
Coming back to `_SAX2DOM`, it is expected to be of type `SAX2DOMMaxExposure` when looking at it from top-down (*DOM* view) from a bottom-up (*SAX* view) it is of its base type `AbstractSAXHandlerMaxExposure`.
It will be produced by an instance of an `SAX2DOMHandlerMaxExposureFactory` extending `AbstractSAXHandlerMaxExposureFactory` provided by a of our library.
`DssXpXMLFilter` is registered with the underlying `SAXSource` and assured via its reference `_CurrentSAXHandlerMaxExposure`, that the actual payload (multiple <dss:InlineXML>, multiple <ds:KeyInfo>, etc ...) are being fed with the *SAX* events, Figure 5.3.6 shows this bottom-up view.

```
Method: ``unmarshalWithSAXPipelineAndSAXExtraction''

  ...
    XMLFilter dssXpXMLFilter = new DssXpXMLFilter(createXMLReader(),
    saxHdlMaxExpFac, dssXpDocumentsStore);


    /* Create a SAX Source to plug it into JAXB */
    SAXSource source = new SAXSource(dssXp, in);
  ...
```

Figure 5.3.5: Shows how the *SAX* level processing is plugged in.



<<datatype>>
XMLFilterImpl
{ From DssXpModelB }
DssXpXMLFilter
{ From sax }
Attributes
package DeclHandler _DeclHandler
package LexicalHandler _LexicalHandler
private String DECLARATION_HANDLER_URI = SAX2Constants.SAX_PROPERTY_PREFIX + SAX2Constants.SAX_PROPERTY_DECLARATION_HANDLER
private String LEXICAL_HANDLER_URI = SAX2Constants.SAX_PROPERTY_PREFIX + SAX2Constants.SAX_PROPERTY_LEXICAL_HANDLER
private int _Opaque
public String DOCUMENT_DSS_ELEMENT = "Document"
private ContentHandler _ContentHandler
private DTDHandler _DTDHandler
private DeclHandler _DeclHandlerParent
private LexicalHandler _LexicalHandlerParent
private String _CurrentKey
public String STRING_INTERNING = SAX2Constants.SAX_FEATURE_PREFIX + SAX2Constants.SAX_FEATURE_STRING_INTERNING
Operations
public DssXpXMLFilter(XMLReader parent, AbstractSAXHandlerMaxExposureFactory abstractSAXHandlerMaxExposureFactory, DssXpDataStore disiDocumentsStore)
public void setProperty(String uri, Object handler)
public void parse(InputSource in)
private void newCurrentSAXHandlerMaxExposure( )
private void returnToParentSAXHandler( )
public void startElement(String uri, String localName, String qName, Attributes atts)
public void endElement(String uri, String localName, String qName)
public void characters(char ch[0..*], int start, int length)
public void startPrefixMapping(String prefix, String uri)
public void endPrefixMapping(String prefix)
public void processingInstruction(String target, String data)
public void skippedEntity(String name)
public void notationDecl(String name, String publicId, String systemId)
public void unparsedEntityDecl(String name, String publicId, String systemId, String notationName)
public void comment(char buf[0..*], int offset, int length)
public void startCDATA( )
public void endCDATA( )
public void startDTD(String name, String publicId, String systemId)
public void endDTD( )
public void startEntity(String name)
public void endEntity(String name)
public void elementDecl(String name, String model)
public void internalEntityDecl(String name, String value)
public void externalEntityDecl(String name, String publicId, String systemId)
public void attributeDecl(String eName, String aName, String type, String valueDefault, String value)
SAXRoundTripHandler
{ From sax }
Attributes
Operations
DssXpDataStore
{ From util }
Attributes
private Map _Map
private int _nullCount
Operations
public DssXpDataStore(Map store)
public Document getXMLData(URI uri)
public Document remove(URI uri)
public Document getXMLData(String uri)
public void put(String refURI, Document xmlDataContents)
_inputDocsStore
Object
{ From dsig }
Attributes
Operations
_SAXHandler
AbstaractSAXHandlerMaxExposure
{ From sax }
Attributes
private ContentHandler _ContentHandler
private LexicalHandler _LexicalHandler
private DTDHandler _DTDHandler
private DeclHandler _DeclHandler
Operations
public Object getResult( )
protected AbstaractSAXHandlerMaxExposure(Object saxHandler)
public Object getSAXHandler( )
public ContentHandler getContentHandler( )
public DTDHandler getDTDHandler( )
public LexicalHandler getLexicalHandler( )
public DeclHandler getDeclHandler( )
_CurrentSAXHandlerMaxExposure
log
Log
{ From misc }
Operations
public void debug(Object msg)
public void debug(Object msg, Throwable e)
public void trace(Object msg)
public void info(Object msg)
public void error(Object msg)
public void error(Object msg, Throwable e)
public void fatal(Object msg)
public void fatal(Object msg, Throwable e)
public void warn(Object msg)
public void warn(Object string, Throwable e)
public boolean isDebugEnabled( )
public boolean isInfoEnabled( )
public void setLevel(short level)
public void setLevel(Class classInfo, short level)
public short getLevel( )
public void setOut(PrintStream out)
public PrintStream getOut( )
log
_SAXHandlerMaxExposureFactory
AbstractSAXHandlerMaxExposureFactory
{ From sax }
Operations
public AbstaractSAXHandlerMaxExposure newSAXHandlerMaxExposure( )
log

Figure 5.3.6: DssXpXMLFilter for *SAX* level extraction or passing all events into the *DOM* binding.

5.4 Implementation - Conclusions[6]

- Architecture, design and implementation for creating an *OASIS-DSS* library prototype have been carried out.
- The reference implementation of the *JAXB API* available in the Java™ Web Service Development Pack (JWSDP) 1.6 has been patched for round-trip support. We have successfully modified the run-time environment for *JAXB* 1.0.5.
- The software library prototype produced for this Master's Thesis has been successfully used in initial interoperability tests, carried out with other participants of the *OASIS-DSS TC*. The software library prototype was also used by another project for an *OASIS-DSS* demo service[80].

[6]See also section 4.4 on page 99.

Appendix A

Appendix

A.1 *XML* in *XML*

```
<?xml version="1.0" encoding="UTF-8"?>
<dss:SignRequest xmlns:dss="urn:oasis:names:tc:dss:1.4142:core:schema">
  <dss:InputDocuments>
    <?lxml <?xml version="1.4142" encoding="UTF-8"+>
      <foo:documentElement xmlns:foo="http://www.example.org/foo">
        Hello World! <![CDATA[ <xml> & </xml> ]]>
      </foo:documentElement>
    <?lxml?>
    <?lxml <?xml version="1.4142" encoding="UTF-8"+>
      <foo:documentElement xmlns:foo="http://www.example.org/foo">
        Hello Universe!
        <?lxml <?xml version="1.4142" encoding="UTF-8"+>
          <bar:documentElement xmlns:foo="http://www.example.org/bar">
            Hello World!
            <?lxml <?xml version="1.4142" encoding="UTF-8"+>
            <?xml version="1.0" encoding="UTF-8"+>
            <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
            <html xmlns="http://www.w3.org/1999/xhtml">
              <head>
                <title>XHTML Document</title>
              </head>
              <body>
                <p id="same">In this World!</p>
              </body>
            </html>
            <?lxml+>
          </bar:documentElement>
        <?lxml+>
      </foo:documentElement>
    <?lxml?>
    <![CDATA[
      <?xml version="1.0" encoding="UTF-8"+>
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
      <html xmlns="http://www.w3.org/1999/xhtml">
        <head>
          <title>XHTML</title>
        </head>
        <body>
          <script type="text/javascript">
            <![CDATA[
            ... unescaped script content ...
            ]]#>
          </script>
          <p id="same">Text</p>
        </body>
      </html>
    ]]>
  </dss:InputDocuments>
</dss:SignRequest>
```

Figure A.1.1: Pitfalls of *XML* in *XML*

In figure Figure A.1.1 we can see various combinations of payload a service like *OASIS-DSS* should be able to cope with. The internal subset of *DTDs* has been left out for keeping the example simple.

The contents of this file should be copied to a COTS *XML* editor supporting syntax highlighting. Then the various combinations of replacing "+" with a "?" and deleting "#" can be tried out.

A.2 *XSLT* to normalise base64

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="1.0">

 <xsl:output indent="no" method="xml" omit-xml-declaration="yes"/>

 <!-- First we need a template for an identity transformation copying all
      nodes to the new document -->
     <xsl:template match="node()|@*">
       <xsl:copy>
         <xsl:apply-templates select="@* | node()"/>
       </xsl:copy>
     </xsl:template>

 <!-- Second, a template for transforming content of type xs:base64Binary
      into a character sequence without linebreaks and whitespaces -->
     <xsl:template
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:dss="urn1:oasis:names:tc:dss:1.0:core:schema"
match="
  //ds:Signature/ds:SignatureValue |
  //ds:Signature/ds:SignedInfo/ds:Reference/ds:DigestValue |
  ...
  //ds:KeyInfo/ds:KeyValue/ds:DSAKeyValue/ds:P |
  //ds:KeyInfo/ds:KeyValue/ds:DSAKeyValue/ds:P |
  //ds:KeyValue/ds:DSAKeyValue/ds:P |
  //ds:KeyValue/ds:DSAKeyValue/ds:Q |
  //ds:KeyValue/ds:DSAKeyValue/ds:G |
  //ds:KeyValue/ds:DSAKeyValue/ds:Y
      "
    >
          <xsl:copy>
            <xsl:value-of select="translate(normalize-space(.),' ','')"/>
          </xsl:copy>
    </xsl:template>

</xsl:stylesheet>
```

Figure A.2.1: *XSLT* to normalise base64

A.3 Recursive *Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<fs:Folder xmlns:fs="http://www.example.com/xml-filesys/" name="A"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/xml-filesys/
     RecursiveSchemaComplexType.xsd">

  <fs:Folder name="MyBinaryFiles">
    <fs:BinaryFile name="x">
    abc=
    </fs:BinaryFile>
  </fs:Folder>
  <fs:Folder name="MyTextFiles">
    <fs:TextFile name="myhtml">
      <html>
        <head>
          <title>title of your HTML document</title>
        </head>
        <body>
          <p>body text of your HTML document</p>
        </body>
      </html>
    </fs:TextFile>
  </fs:Folder>

</fs:Folder>
```

Figure A.3.1: instance document for recursive *Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fs="http://www.example.com/xml-filesys/"
  targetNamespace="http://www.example.com/xml-filesys/"
  >

  <xs:simpleType name="folderAndFileNameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
      <xs:pattern value="[A-Za-z].*"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="Folder" type="fs:FolderType"/>
  <xs:complexType name="FolderType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="fs:Folder"/>
      <xs:element ref="fs:TextFile"/>
      <xs:element ref="fs:BinaryFile"/>
    </xs:choice>
    <xs:attribute name="name" type="fs:folderAndFileNameType"/>
  </xs:complexType>
  <xs:element name="TextFile" type="fs:TextFileType"/>
  <xs:complexType name="TextFileType" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:any namespace="##local" processContents="lax"/>
      <xs:any namespace="##other" processContents="lax"/>
      <xs:any namespace="##targetNamespace" processContents="strict"/>
    </xs:choice>
    <xs:attribute name="name" type="fs:folderAndFileNameType"
        use="required"/>
  </xs:complexType>

  <xs:element name="BinaryFile" type="fs:BinaryFileType"/>
  <xs:complexType name="BinaryFileType">
    <xs:simpleContent>
      <xs:extension base="xs:base64Binary">
        <xs:attribute name="name" type="fs:folderAndFileNameType"
            use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

</xs:schema>
```

Figure A.3.2: recursive *Schema*

A.4 Example of a ValidateType

```
<xs:complexType name="ValidateType" abstract="true">
  <xs:sequence>
    <xs:element name="GrammarLanguage" type="xs:anyURI"/>
    <xs:element name="Hint" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
          <xs:sequence>
          <xs:any minOccurs="0"/>
          </xs:sequence>
        <xs:attribute name="namespace" type="xs:anyURI" use="optional"/>
        <xs:attribute name="hint" type="xs:NCName" use="optional"/>
        <xs:attribute name="value" type="xs:anyURI" use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figure A.4.1: Example of a ValidateType

A.5 An Example of a very complex XML document.

```
<?xml␣version="1.0"␣encoding="UTF-8"?>
␣␣␣␣␣␣<!--␣comments␣followed␣by␣insignificant␣whitespace
␣␣␣␣␣␣␣␣␣processing␣instructions␣-->
␣␣␣<?ProcessingInstructionTarget␣content?>
<!DOCTYPE␣documentElement␣[
␣␣<!ELEMENT␣documentElement␣(#PCDATA␣|␣firstChild␣|␣secondChild)*>
␣␣<!ELEMENT␣firstChild␣(#PCDATA)>
␣␣<!ELEMENT␣secondChild␣(#PCDATA)>
]>
␣␣␣␣<!--␣more␣comments-->
␣␣<documentElement>␣A␣man␣has␣two␣children,␣one␣is␣called
␣␣␣␣␣<firstChild>␣Alice
␣␣␣␣␣</firstChild␣␣>␣and␣the␣other␣one␣is␣called
<secondChild>␣Bob␣</secondChild>.
␣␣␣␣␣␣</documentElement>
␣␣␣␣<!--␣more␣comments␣-->

␣␣␣<?ProcessingInstructionTarget␣content?>

␣␣␣␣␣␣␣<!--␣more␣comments␣-->
```

Well-formed complex *XML* document, with *processing instructions comments* and insignificant whitespace, before and after the document element and a *mixed content* model.

Figure A.5.1: Well-formed complex *XML* document.

A.6 XML Derivatives and Alternatives

Most of the information in this section is derived from a web page by someone who calls himself PaulT [1] `http://www.pault.com/xmlalternatives.html` [2]. A shortened and edited version, where XML shorthands and subsets have been left out intentionally follows:

Name	Short Description	Language
lml	Lambda Markup Language, William D. Lindsey's Stupid Net Tricks proposal.	
SEXP	S-expressions are a variation on LISP S-expressions by Ron Rivest	C
sfsexp	small, fast s-expression library lighter weight than the Rivest s-expression parser	C/C++
SXML	SXML is an abstract syntax tree of an XML document. SXML is also a concrete representation of the *Infoset* in the form of S-expressions	Scheme

Figure A.6.1: XML Alternatives based on LISP-like or Scheme-like Expressions

[1]PauT may be Paul Tchistopolskii the co-author of "'Professional Xsl"' and version 0.1 of a web server called Hiawatha

[2]Accessed via a web archive `http://web.archive.org/web/20060409100013/http://www.pault.com/xmlalternatives.html`

Name	Short Description	Language
Boulder	hierarchical name=value structures	Perl,Java
config	JSON-like syntax, allows cross-referencing, imports	Python
GODDAG	General Ordered-Descendant Directed Acyclic Graph allows overlapping hierarchies in a plausible data structures for representing documents with overlap	
HDF	Hierarchical Data Format, XML and XSLT alternative	C, Java, Perl, Python
JSON	JavaScript Object Notation (JSON), lightweight data-interchange format	Java, C, C++, ...
OGDL	Ordered Graph Data Language, uses indention, nodes are strings, arcs or edges are spaces or indentation	C, Perl, Java
ONX	text-based, no *mixed content*, can contain binary	C++
SDL	Simple Data Language, elements, attributes, designed for configuration	Java
SMEL	inspired by XML, has elements, values, comments and directives.	Euphoria
SSYN	Structured SYntax is intended to be a simpler alternative to data-centric XML and YAML	Python
XML plist	XML plist significant in Mac OS X. Property lists come from NeXTStep (OS from NeXT 1989, founder Steve Jobs)	Objective-C, Java, C
YAML	”YAML Ain’t Markup Language”(recursive acronym), uses indention, human readable, chiefly for scripting languages	JavaScript, Perl, Python, Ruby, C

Figure A.6.2: XML Alternatives

Name	Short Description	Language
APT	Almost Plain Text, uses paragraph indentation for simple article-like documents	
Markdown	Markdown text to valid XHTML (or HTML)	Perl
txt2tags	txt2tags converts text to XHTML, LaTeX, moinmoin, man pages and others	Python
WikiMl	Wiki text as Markup Language equivalent, transform to XHTML, SAX events, XSLT transformations	Java

Figure A.6.3: XML Alternatives mainly for simple text2html publishing

A.6.1 Bug in *Xalan*

```
<xsl:variable name="precedingLeafNodes"
     select="count( preceding::node()[not(child::node())])"/>
S
<xsl:variable name="precedingLeafNodes"
  select="count(
    ancestor-or-self::node()/
     preceding-sibling::node()/
      descendant-or-self::node()
        [not(child::node())]
  )"/>
```

Figure A.6.4: *Xalan* returns different results for those, why?

A.6.2 Bug in *JAXB*

“`http://www.nabble.com/BUG-in-DefaultJAXBContextImpl.newInstance(java.lang.Class-javaContentInterface)-td6287291.html#a6287291`”

Bibliography

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[2] OASIS. FAQ, OASIS Digital Signature Services TC. “`http://www.oasis-open.org/committees/dss/faq.php`”. accessed December 2nd 2004.

[3] OASIS. Charter, OASIS Digital Signature Services TC. “`http://www.oasis-open.org/committees/dss/charter.php`”. accessed December 1st 2004.

[4] ETSI. XML format for signature policies. “`http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=13350`”, April 2002. accessed March 28th 2008.

[5] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). “`http://www.w3.org/TR/2008/REC-xml-20081126/`”, November 2008.

[7] James Clark. Comparison of SGML and XML. “`http://www.w3.org/TR/NOTE-sgml-xml-971215`”, December 1997.

[8] Karl Scheibelhofer. Signing XML Documents and the Concept of What You See Is What You Sign. Master’s thesis, IAIK, Graz University of Technology, January 2001. “`http://www.iaik.tugraz.at/teaching/11_diplomarbeiten/archive/scheibelhofer.pdf`”.

[9] Elliotte Rusty Harold. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley, November 2002. “`http://www.cafeconleche.org/books/xmljava/`”.

[10] David Orchard. Extensibility, XML Vocabularies, and XML Schema. “`http://www.xml.com/pub/a/2004/10/27/extend.html`”, October 2004. accessed 1st of Febuary 2005.

[11] Donald E. Eastlake III and Kitty Niles. *Secure XML, The New Syntax for Signatures and Encryption*. Addison Wesley, 2003.

[12] Kurt Cagle, John Duckett, Oliver Griffin, Stephen Mohr, Francis Norton, Nikola Ozu, Ian Stokes-Rees, Jeni Tennison, and Kevin Williams. *Professional XML Schemas*. Wrox Press, 2003.

[13] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly & Associates, second edition edition, June 2002.

[14] Elliotte Rusty Harold. *Effective XML: 50 Specific Ways to Improve Your XML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. "`http://www.cafeconleche.org/books/effectivexml/`".

[15] Jonathan Marsh and Joanne Tong. XML Path Language (XPath) Version 1.0 Specification Errata. "`http://www.w3.org/1999/11/REC-xpath-19991116-errata/`", November 2005. accessed December 16th 2008.

[16] Steven Murdoch. Survey of general-purpose data-representation formats and markup languages. "`http://www.cl.cam.ac.uk/~sjm217/projects/markup/survey/`". accessed 4th of May 2007.

[17] Gregor Karlinger. Digital Signatures in XML. An Implementation in Java. Master's thesis, IAIK, Graz University of Technology, January 2000. Written in German and not availiable online.

[18] Michael Kay. *XPath 2.0 Programmer's Reference*. Wiley, 2004.

[19] Dennis M. Sosnoski. XML and Java technologies: Data binding, Part 1: Code generation approaches – JAXB and more. "'`http://www-106.ibm.com/developerworks/library/x-databdopt/index.html`"'. accessed 20th of January 2005.

[20] W3C. XML Schema Second Edition. Part-0 Primer: "`http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`", Part-1 Structures: "`http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`", Part-2 Datatypes: "`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`". accessed March 17th 2005.

[21] Eric van der Vlist. *XML Schema*. O'Reilly & Associates, first edition, german translation edition, January 2003.

[22] Roger L. Costello et al. XML Schemas: Best Practices - Global versus Local - A Collectively Developed Set of Schema Design Guidelines. "`http://web.archive.org/web/20010405144847/www.xfront.com/BestPractices.html`", April 2001. "`http://web.archive.org/web/20010405154801/www.xfront.com/GlobalVersusLocal.html`".

[23] David Brownell. *SAX2*. O'Reilly & Associates, first edition edition, January 2002.

[24] Jonathan Marsh, Daniel Veillard, and Norman Walsh. xml:id Version 1.0. "`http://www.w3.org/TR/xml-id/`", September 2005.

[25] Tim Berners-Lee, Ron Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. “`http://tools.ietf.org/html/rfc3986`”, January 2005. obsoletes: RFC1808, RFC2732, RFC2396.

[26] Tim Berners-Lee, Ron Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. “`http://tools.ietf.org/html/rfc2396`”, August 1998. obsoleted by RFC3986[25].

[27] W3C. XML-Signature Syntax and Processing. “`http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`”, 2002. Recommendation, accessed December 10th 2004.

[28] R. Hinden, B. Carpenter, and Larry Masinter. Format for Literal IPv6 Addresses in URL’s. “`http://tools.ietf.org/html/rfc2732`”, December 1999. obsoleted by RFC3986[25].

[29] W3C. XML Signature Syntax and Processing (Second Edition). “`http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/`”, March 2008. Proposed Edited Recommendation, accessed May 5th 2008.

[30] W3C. XPointer Framework. “`http://www.w3.org/TR/2003/REC-xptr-framework-20030325/`”, March 2003. Recommendation, accessed December 10th 2007.

[31] Steven DeRose, Eve Maler, and Ron Daniel Jr. XML Pointer Language (XPointer) Version 1.0 W3C Candidate Recommendation. “`http://www.w3.org/TR/2001/CR-xptr-20010911/`”, 2001. accessed December 10th 2007.

[32] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. “`http://www.w3.org/TR/xpath`”, November 1999. accessed December 10th 2007.

[33] Steve DeRose, R. Daniel, Eve Maler, and Jonathan Marsh. XPointer xmlns() Scheme. W3C Recommendation. “`http://www.w3.org/TR/2003/REC-xptr-xmlns-20030325/`”, March 2003. accessed December 10th 2007.

[34] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. XPointer element() Scheme. W3C Recommendation. “`http://www.w3.org/TR/2003/REC-xptr-element-20030325/`”, March 2003. accessed December 10th 2007.

[35] Steve DeRose, R. Daniel, and Eve Maler. XPointer xpointer() Scheme. W3C Working Draft. “`http://www.w3.org/TR/2002/WD-xptr-xpointer-20021219/`”, December 2002. accessed December 10th 2007.

[36] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. “`http://www.w3.org/TR/webarch/`”, December 2004. accessed August 1st 2008.

[37] W3C. XPointer Implementations. “`http://www.w3.org/XML/2002/10/LinkingImplementations.html`”, 2002. accessed May 5th 2008.

[38] Leigh Dodds. XPointer and the Patent. “`http://www.xml.com/pub/a/2001/01/17/xpointer.html`”, January 2001. accessed 1st of Febuary 2005.

[39] Arno Hollosi, Gregor Karlinger, Thomas Rssler, Martin Centner, and al. Die sterreichische Brgerkarte. “`http://www.buergerkarte.at/konzept/securitylayer/spezifikation/20080220/`”, February 2008.

[40] Bradley W. Hill. Command Injection in XML Signatures and Encryption. “`http://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf`”, July 2007.

[41] Hakon W. Lie and Bert Bos. *Cascading Style Sheets*. Pearson Addison Wesley, June 2004.

[42] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001. “`http://www.win.tue.nl/~klenstra/key.pdf`”.

[43] Konrad Lanz, Dieter Bratko, and Peter Lipp. RSA-PSS in XMLDSig. “`http://www.w3.org/2007/xmlsec/ws/papers/08-lanz-iaik/`”, September 2007.

[44] Florian Mendel, Christian Rechberger, and Vincent Rijmen. Update on SHA-1. “`http://www.iaik.tugraz.at/aboutus/people/rechberger/talks/Rechberger_SHA1BOINC_V07.pdf`”, January 2007. accessed May 5th 2008.

[45] Florian Mendel, Christian Rechberger, and Vincent Rijmen. New SHA-1 Collision Attacks, and Applications. “`http://wiki.uni.lu/esc/docs/NewSHA1CollisionSearch.pdf`”, January 2008. accessed May 5th 2008.

[46] ETSI. XML Advanced Electronic Signatures (XAdES). “`http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=21353`”, March 2006. TS 101 903, accessed March 28th 2008.

[47] ETSI. CMS Advanced Electronic Signatures (CAdES) . “`http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=28069`”, November 2007. TS 101 733, accessed May 5th 2008.

[48] Martin Centner. XML Advanced Electronic Signatures (XAdES) - Implementation and Interoperability. Master’s thesis, IAIK, Graz University of Technology, September 2003. “`http://www.iaik.tu-graz.ac.at/teaching/11_diplomarbeiten/archive/mcentner.pdf`”.

[49] John Boyer, Merlin Hughes, and Joseph Reagle. XML-Signature XPath Filter 2.0 W3C Recommendation. "`http://www.w3.org/TR/xmldsig-filter2/`", November 2002. accessed July 5th 2008.

[50] D. Eastlake 3rd, J. Reagle, and D. Solo. XML-Signature Syntax and Processing. "`http://tools.ietf.org/html/rfc3075`", March 2001. Obsoleted by [57].

[51] D. Eastlake 3rd. Additional XML Security Uniform Resource Identifiers (URIs). "`http://tools.ietf.org/html/rfc4051`", April 2005. Errata: "`http://www.rfc-editor.org/errata_search.php?rfc=4051`", likely to be obsoleted by "`http://tools.ietf.org/html/draft-eastlake-additional-xmlsec-uris-00`".

[52] John Boyer. Canonical XML Version 1.0 W3C Recommendation. "`http://www.w3.org/TR/xml-c14n`", March 2001. accessed December 12th 2004.

[53] John Boyer and Glenn Marcy. Canonical XML Version 1.1 W3C Recommendation. "`http://www.w3.org/TR/xml-c14n11`", March 2001. contributions by Konrad Lanz [54], accessed December 20th 2006.

[54] Konrad Lanz and José Kahan. Known Issues with Canonical XML 1.0 (C14N/1.0) W3C Working Group Note. "`http://www.w3.org/TR/C14N-issues/`", December 2006. accessed December 12th 2004.

[55] John Boyer, Donald E. Eastlake 3rd, and Joseph Reagle. Exclusive XML Canonicalization Version 1.0 W3C Recommendation. "`http://www.w3.org/TR/xml-exc-c14n/`", July 2002. accessed December 15th 2004.

[56] Bob Atkinson. Schema Centric XML Canonicalization Version 1.0. "`http://uddi.org/pubs/SchemaCentricCanonicalization-20050523.htm`", May 2001. accessed December 10th 2005.

[57] D. Eastlake 3rd, J. Reagle, and D. Solo. XML-Signature Syntax and Processing. "`http://tools.ietf.org/html/rfc3275`", March 2002. Obsoletes by [50].

[58] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0 (Second Edition). "`http://www.w3.org/TR/2006/REC-xml-names-20060816/`", August 2006.

[59] Norman Walsh. Using Qualified Names (QNames) as Identifiers in XML Content. "`http://www.w3.org/2001/tag/doc/qnameids`", March 2004. accessed September 22nd 2008.

[60] DSS Overview, Working Draft 04. "`http://www.oasis-open.org/committees/download.php/20051/oasis-dss-1.0-overview-wd-04.doc`", August 2005. contribution by Konrad Lanz, accessed August 29th 2005.

[61] OASIS Open. Advanced Electronic Signature Profiles of the OASIS Digital Signature Service Version 1.0. “`http://docs.oasis-open.org/dss/v1.0/oasis-dss-profiles-AdES-spec-v1.0-os.html`”, April 2007. accessed Febuary 10th 2008.

[62] OASIS. Digital Signature Service Core Protocols, Elements, and Bindings Version 1.0 (OASIS Standard), 11 April 2007. “`http://www.oasis-open.org/committees/dss`”, April 2007. accessed Febuary 10th 2008.

[63] OASIS. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). “`http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`”, February 2006.

[64] Adam Bosworth, Don Box, Martin Gudgin, Mark Nottingham, David Orchard, and Jeffrey Schlimmer. XML, SOAP and Binary Data. “`http://www.xml.com/pub/a/2003/02/26/binaryxml.html`”, February 2003. accessed April 12th 2005.

[65] Tim Berners-Lee. Web Architecture from 50,000 feet. “`http://www.w3.org/DesignIssues/Architecture.html`”, September 1998. updated October 1999, cvs dated February 2002.

[66] Norman Walsh. XML 2.0. “`http://norman.walsh.name/2004/11/10/xml20`”, November 2004. modified 11 Sep 2005.

[67] Tim Bray. Extensible Markup Language - SW (XML-SW). “`http://www.textuality.com/xml/xmlSW.html`”, February 2002. retrieved via “`http://web.archive.org/web/*/www.textuality.com/xml/xmlSW.html`”.

[68] Norman Walsh. XML 2.0? No, seriously. “`http://norman.walsh.name/2008/02/20/xml20`”, February 2008. modified 09 May 2008.

[69] Bradley W. Hill. Attacking XML Security. `http://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf`”, July 2007.

[70] Christian Geuer-Pollmann. Anwender-Probleme mit XML Signature. “`http://www.nue.et-inf.uni-siegen.de/~geuer-pollmann/publications/20030403_XMLSignaturWorkshop/`”, April 2003. retrieved via “`http://web.archive.org/web/*/www.nue.et-inf.uni-siegen.de/~geuer-pollmann/publications/20030403_XMLSignaturWorkshop/`”.

[71] Joseph Reagle. XML Validation Transforms for XML Signature W3C Working Draft. “`http://www.w3.org/Signature/Drafts/xmldsig-transform-xml-validation.html`”, October 2001. accessed September 22nd 2008.

[72] Joseph Reagle Jr. XML-Signature Requirements, W3C Working Draft . "`http://www.w3.org/TR/xmldsig-requirements`", October 1999.

[73] Tim Berners-Lee. Principles of Design. "`http://www.w3.org/DesignIssues/Principles.html`", September 1998. last change:16th January 2008.

[74] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.1 (Second Edition). "`http://www.w3.org/TR/2006/REC-xml-names11-20060816/`", August 2006.

[75] Michael McIntosh and Paula Austel. XML Signature Element Wrapping Attacks and Countermeasures. "`http://domino.research.ibm.com/library/cyberdig.nsf/papers/73053F26BFE5D1D385257067004CFD80/`", August 2005.

[76] Sebastian Gajek, Lijun Liao, and Jrg Schwenk. Towards a Semantic of XML Signature. "`http://www.w3.org/2007/xmlsec/ws/papers/07-gajek-rub/`", September 2007.

[77] Jeff Hodges and Scott Cantor. XML Canonicalization: The Weakest Link. "`http://www.w3.org/2007/xmlsec/ws/papers/06-zhang-ximpleware/`", September 2007.

[78] Sekhar Vajjhala and Joe Fialli. The Java™ Architecture for XML Binding (JAXB) - Final, V1.0. "`http://jcp.org/en/jsr/detail?id=31`", January 2003.

[79] Kohsuke Kawaguchi, Sekhar Vajjhala, and Joe Fialli. The Java™ Architecture for XML Binding (JAXB) 2.1 - Final Release. "`http://jcp.org/en/jsr/detail?id=222`", December 2006.

[80] Bernd Zwattendorfer and Thomas Zefferer. PROJEKTBERICHT - OASIS-DSS DIGITAL SIGNATURE SERVICE. "`http://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf`", December 2007. "`http://demo.a-sit.at/el_signatur/dss/index.html`".

[81] Clements et al. *Documenting Software Architectures*. Addison Wesley. ISBN 0201703726.

[82] Sun. Web Services - Praktikum aus Softwareentwicklung 2. "`http://java.sun.com/webservices/docs/1.4/tutorial/doc/JAXBUsing4.html`". accessed 27th of January 2005.

[83] Sun. JavaTM Architecture for XML Binding, JAXB RI Vendor Extensions Customizations. "`http://java.sun.com/webservices/docs/1.5/jaxb/vendorCustomizations.html#dom`". accessed 1st of Febuary 2005.

[84] Jonathan Marsh. XML Base. "`http://www.w3.org/TR/xmlbase/`", June 2001.

Glossary

acronym (*ACR*)

Acronyms written in italics can be found in this glossary.* page(s) viii, 1, 6, 157

italics

Terms written in italics in the text can be found in this glossary.* page(s) 1, 6, 157

anonymous type

An anonymous type in *Schema* is either a *simple type* or *complex type* that is specified in-line and has local scope. The elements of local types are affected by the `elementFormDefault` attribute.
`http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#Element_Declaration_details*` page(s) 11, 54, 55, 159

Application Program Interface (API)

Abbreviation of application program interface, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer puts the blocks together.
`http://www.webopedia.com/TERM/A/API.html*` page(s) 8, 14–16, 18, 24, 35, 37, 60, 67, 78, 91, 93, 99, 102, 111, 118, 149, 157

AbstractSyntaxNotationOne (ASN.1)

Abstract Syntax Notation One is used to define abstract data types and formats. `http://asn1.elibel.tm.fr/en/introduction/index.htm*` page(s) 8, 138

Apache Axis

Apache Axis is Web service framework and XML based. It is an open source *SOAP* server.
`http://www.axis*` page(s) 87

base URI reference

The *URI* of the containing entity, like the file or an element.

- section 4.4 RFC3986[25]
- section 4.2 RFC2396[26]

* page(s) 149, 157

Base64 Content-Transfer-Encoding (base64 encoding)

A subset of US-ASCII is used to encode arbitrary binary data using 64 characters and encodes 6 bits per character. It is used to encode binary data in text formats like XML.
`http://tools.ietf.org/html/rfc2045#section-6.8`
`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#base64Binary`*
page(s) 57, 58, 61, 69, 70, 77, 78, 80, 82, 83, 86, 99, 114

Backus Naur Form (BNF)

The BackusNaur Form (BNF) defines a syntax for expressing context-free grammars.* page(s) 11, 31, 37, 77, 153

CMS Advanced Electronic Signatures (CAdES)

CMS Advanced Electronic Signatures (CAdES) defines various properties for CMS Signatures encoded in *ASN.1*. These properties range from the signing time and signature policy over the signing certificate to properties needed for long term verification and archival of electronic signatures. [47] `http://tools.ietf.org/html/rfc5126`* page(s) 36, 53, 74

Canonical XML Version 1.0 W3C Recommendation 15 March 2001

Converts an *XPath* node-set into an octet stream that has a canonical form[52]. The degree normalization that is achieved is however limited especially when it comes to namespaces. See subsection 2.5.2.* page(s) 2, 15, 18, 22, 41–45, 47–49, 62, 63, 67, 78, 82, 84, 86, 90, 93–95, 97, 110, 138, 150, 153–155, 157

Canonical XML Version 1.1 W3C Recommendation

Converts an *XPath* node-set into an octet stream that has a canonical form. *C14n11* is the successor of *C14n*and fixes problems with *xml:id* and *xml:base* [54] [53]. See subsection 2.5.2.* page(s) 42–44, 46, 49, 88, 90, 93, 94, 110, 138, 150, 153, 155

CDATA sections

A special section `<![CDATA[ ... ]]>` that can contain unescaped text including verbatim markup.
`http://www.w3.org/TR/xml/#sec-cdata-sect`* page(s) 15, 16, 18, 158

Cryptographic Message Syntax (CMS)

Cryptographic Message Syntax (CMS) is a signature format defined by the *IETF* and is based on PKCS#7.
`http://tools.ietf.org/html/rfc3369`
`http://tools.ietf.org/html/rfc3370*` page(s) 39, 53, 138

comment

An *XML* comment `<!-- comment -->`, see expression 2.1.7.
`http://www.w3.org/TR/xml/#sec-comments*` page(s) 4, 6, 15, 18, 22, 33, 41, 95, 98, 113–115, 124, 157

complex type

A complex type in *Schema* can have attributes and elements and specify the order and quantity of the latter by means of sequences and choices
`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#rf-defn*` page(s) 12, 55, 137, 158, 159

Crimson

Crimson was the parser preceding *Xerces* in SUN's JDK.
`http://xml.apache.org/crimson/*` page(s) 16, 157

Certificate Revocation List (CRL)

A Certificate Revocation List (CRL) is a signed list of references (serial numbers) to certificates that have been revoked. `http://tools.ietf.org/html/rfc3280*` page(s) 51, 157

Computer supported cooperative work (CSCW)

Computer supported cooperative work (CSCW) is a research domain covering technology supporting inter human interaction. The CSCW Matrix was introduced in 1988 by Johansen.* page(s) 2

Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a style sheet language that allows authors and users to attach style to structured documents (e.g., HTML documents and XML applications). By separating the presentation style of documents from the content of documents, CSS simplifies Web authoring and site maintenance.
`http://www.w3.org/Style/CSS/`
`http://www.w3.org/TR/CSS21/*` page(s) 4, 15, 35

Data Flow Diagram (DFD)

The Data Flow Diagram allows to show a system as functions or processes represented by circles that are connected via streams or pipelines represented as arcs between them and should be mostly self explaining. Two lines represent a data store. `http://www.yourdon.com/jesa/jesa.php`* page(s) 37, 41, 153

Document Type Declaration (DocType)

The W3C's Definition: The XML document type declaration contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together. `http://www.w3.org/TR/REC-xml/#dt-doctype`* page(s) 5, 6, 17, 85, 97, 153

document element

The document element is the top most element in an *XML* document and should not be confused with the document root as specified in the *XPath* data model which is at the top of a document. Hence the term root element should be avoided.
`http://www.w3.org/TR/xml/#dt-root`* page(s) 5, 6, 16, 18, 22, 43, 44, 97, 145, 157

document node

The document node in *DOM* corresponds to the *root node* in *XPath* and to an *XML* file/stream.* page(s) 17, 18, 146, 157

document order

The document order, is the order of nodes in an *XML* document determined by the appearance of the first character within *XML's* serial form. Attribute nodes are not counted and are conceptually sets that hang laterally off their owning element node. `http://www.w3.org/TR/xpath#dt-document-order`* page(s) 17, 59, 110, 111, 157

Document Object Model (DOM)

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.
`http://www.w3.org/DOM/`
`http://www.w3.org/TR/DOM-Level-2-Core/`* page(s) 4, 14–18, 44, 77, 78, 94, 96, 98, 110–113, 115–117, 140, 146, 149, 155

OASIS Digital Signature Services (DSS)

OASIS Digital Signature Services (DSS) is a client-server request response protocol for signature creation, verification and time-stamping. See chapter 3.* page(s) 1–3, 5, 8, 9, 11, 13, 15, 23–25, 27, 28, 35, 51–54, 57–68, 70, 72, 74–78, 80, 84, 87–89, 95–98, 101–105, 110, 111, 113, 114, 116, 118, 120, 141, 153, 154, 157

Digital Signature Standard (DSS)

This Standard specifies a Digital Signature Algorithm (DSA) appropriate for applications requiring a digital rather than written signature.
`http://www.itl.nist.gov/fipspubs/fip186.htm`* page(s) 2

OASIS Digital Signature Services eXtended (DSS-X)

OASIS-DSS-X is the successor *TC* of the *OASIS-DSS TC* and is concerned with maintaining *OASIS-DSS* and specifying new profiles.* page(s) 52, 141

Digital Signature Services XML processor (DssXp)

The Digital Signature Services XML processor (DssXp) is a library that implements the *OASIS-DSS* protocol and supports the creation and verification of *XMLDSIG* signatures.* page(s) 101

Document Type Definition (DTD)

The Document Type Definition is the grammar specified by the markup declarations for a class of documents. The document type declaration (DOCTYPE) may contain internal and also refer to external markup declarations. The DTD is then comprised of the union of internal and external markup declarations. (see also DOCTYPE) `http://www.w3.org/TR/REC-xml/#dt-markupdecl`* page(s) 2, 5, 6, 11, 13, 15, 36, 42–44, 47, 84–86, 97, 120, 154, 157

Electronic Banking Internet Communication (EBICS)

The Zentraler Kreditausschuss (ZKA) in Germany enhanced the Banking Communication Standard (BCS) processes to make them suitable for Internet-based use. It claims to cover credit transfers, direct debits and the whole spectrum of the corporate customers cash management. `http://www.ebics-zka.de/english/index.htm`* page(s) 31

European Telecommunications Standards Institute (ETSI)

The European Telecommunications Standards Institute (ETSI) is an independent, non-profit organization, whose mission is to produce telecommunications standards for today and for the future. `http://www.etsi.org/`* page(s) 9, 53

Exclusive XML Canonicalization Version 1.0 W3C Recommendation 18 July 2002

Converts an *XPath* node-set into an octet stream that has a canonical form avoiding to inherit namespaces and inheritable attributes. This however only has an effect on the `DigestInput` if used in *XMLDSIG*. See subsection 2.5.3.* page(s) 42, 44–49, 57, 58, 90, 95–98, 110, 113, 150, 153, 154

Efficient XML Interchange (EXI) Format 1.0

The Efficient XML Interchange (EXI) format is a very compact representation for *Infoset* intended to optimize performance and utilization of computational resources.
`http://www.w3.org/TR/exi/*` page(s) 80

Keyed-Hash Message Authentication Code (HMAC)

HMAC is a symmetric signature or message authentication method using a shared secret for authentication. `http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf` `http://tools.ietf.org/html/rfc2104*` page(s) 35, 39

Hyper Text Markup Language (HTML)

The HyperText Markup Language (HTML) is the publishing language of the World Wide Web.
`http://www.w3.org/TR/html401/*` page(s) 5, 32, 35, 82, 149

Hyper Text Transfer Protocol (HTTP)

The Hyper Text Transfer Protocol (HTTP) is an application-level (Layer 5) stateless request response protocol. `http://tools.ietf.org/html/rfc2616*` page(s) 53, 68, 80

Institute for Applied Information Processing and Communications (IAIK)

The abbreviation comes from its German name 'Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie' at the Graz University of Technology (TU-Graz) and is active in the areas:
Applied research in computer networking, embedded systems, system-on-chip design, computer security and information security.
Observing rapidly-evolving technologies like information security, networking, and system-on-chip design.
Teaching students in project-oriented work following an inter-disciplinary approach.
`http://www.iaik.tugraz.at/*` page(s) ii, vii, 15, 103, 151

The Internet Engineering Task Force (IETF)

The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet archi-

tecture and the smooth operation of the Internet. It is open to any interested individual.
`http://www.ietf.org/*` page(s) 36, 138

XML Information Set (Infoset)

The *Infoset*provides a set of definitions to descibe the information in an XML document. The main intent is to be used in other specifications. Some specifications like *ScC14n* also makes use of the *Infoset* as a data-model.
`http://www.w3.org/XML/*` page(s) 47, 62, 96, 98, 114, 125, 142, 143, 145, 146, 150, 157

Internationalized Resource Identifiers (IRI)

Internationalized Resource Identifier (IRI) are an extension of Uniform Resource Identifier (URI) about characters from the Universal Character Set (Unicode/ISO 10646). A mapping from IRIs to URIs is defined. `http://tools.ietf.org/html/rfc3987*` page(s) 32, 143, 157

Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) provides a way to bind an XML schema to a representation in Java code. This enables to incorporate XML data and processing functions in applications based on Java technology without having to deal with a lot of XML's idiosyncrasies.
`http://java.sun.com/xml/jaxb/*` page(s) 16, 98, 102, 103, 108, 110–115, 118, 127, 157

Java API for XML Processing (JAXP)

The Java API for XML Processing (JAXP) enables applications to parse and transform XML documents independent of a particular XML processing implementation.
`http://java.sun.com/webservices/jaxp/*` page(s) 15, 18, 32, 105

Java Specification Request 105: XML Digital Signature APIs (JSR105)

JSR105 defines high-level APIs for *XMLDSIG*. `http://jcp.org/en/jsr/detail?id=105*` page(s) 2, 28, 37, 41, 59–61, 67, 91, 93, 102, 104, 105, 110, 111, 151

Legacy extended IRIs for XML resource identification (IRI)

A LEIRI is allowed to have spaces, delimiters and some other characters, disallowed in *IRIs* in their non percent-encoded form. A mapping to *IRIs* is defined. `http://www.w3.org/TR/2008/NOTE-leiri-20081103/*` page(s) 32, 157

MAY

RFC 2119 MAY: Used to identify truly optional items.
`http://tools.ietf.org/html/rfc2119#section-5*` page(s) 59, 157

Minimal Canonicalization (MC14n) PROPOSED STANDARD Obsoleted by: RFC3275[57]

Minimal Canonicalization is a character level text canonicalization defined in RFC3075[50], obsoleted by RFC3275[57] which does not contain it anymore.
`http://tools.ietf.org/html/rfc3075#section-6.5.1*` page(s) 42, 43, 49, 58, 59

mixed content

We talk about mixed content, if an element in an *XML* document has text and element children.
`http://www.w3.org/TR/xml/#sec-mixed-content*` page(s) 4, 6, 40, 43, 55, 68, 84, 96, 97, 124, 125, 157

Organization for the Advancement of Structured Information Standards (OASIS)

The Organization for the Advancement of Structured Information Standards is a not-for-profit, international consortium that drives the development, convergence, and adoption of e-business standards. The consortium produces more Web services standards than any other organization along with standards for security, e-business, and standardization efforts in the public sector and for application-specific markets. Founded in 1993, OASIS has more than 4,000 participants representing over 600 organizations and individual members in 100 countries.
`http://www.oasis-open.org/`
`http://en.wikipedia.org/wiki/OASIS_(organization)*` page(s) 1, 9, 47, 146, 147

Online Certificate Status Protocol (OCSP)

The Online Certificate Status Protocol (OCSP) is used to retrieve the revocation state of a Certificate. `http://tools.ietf.org/html/rfc2560*` page(s) 51

Portable Document Format (PDF)

The Adobe Portable Document Format is the native file format of the Adobe Acrobat family of products and relies on the same imaging model as the PostScript page description language. PDF also includes objects, hyperlinks and other features. `http://www.adobe.com/devnet/pdf/pdf_reference.html*` page(s) 5, 9, 32

Public Key Cryptography Standards (PKCS)

Public Key Cryptography Standards is set of de facto standards issued by RSA Laboratories.
`http://www.rsa.com/rsalabs/node.asp?id=2124*` page(s) 35

Public-Key Infrastructure (PKI)

Public-Keys are bound to real entities by a Certification Authority (CA) aka. Trusted Third Party issuing certificates and revocation information according to the ITU X.509 specifications. X.509:

`http://www.itu.int/rec/T-REC-X.509-200508-I/en*` page(s) vii, 35, 39, 53, 73, 89, 103, 158

prefixed name

A prefixed name is a *QName* composed of a prefix and a local name separated by a colon.
`http://www.w3.org/TR/2006/REC-xml-names-20060816/#NT-PrefixedName*`
page(s) 10, 11, 45, 46, 145, 158

processing instruction

An *XML* processing instruction `<?target content?>`, see expression 2.1.8.
`http://www.w3.org/TR/xml/#sec-comments*` page(s) 4, 6, 15, 17, 18, 33, 35, 78, 98, 113–115, 124, 158

prolog

The prolog is the beginning of an *XML* document and it ends with the *document element*.
`http://www.w3.org/TR/xml/#NT-prolog*` page(s) 5, 6, 40, 78–80, 97, 114, 149, 158

properly nested

Structures in an *XML* document are properly nested, so no element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.
`http://www.w3.org/TR/xml/#sec-documents`
`http://www.w3.org/TR/xml/#wf-entities*` page(s) 4, 158

Post Schema Validation Infoset (PSVI)

The Post Schema Validation *Infoset* is the *Infoset* of the well-formed *XML* plus any additional information derived from a *Schema* during asessment (validation). Things like attributes of type `xs:ID` or default attributes are added.
`http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#key-psvi`
`*` page(s) 13–15, 84, 97, 158

public identifier

The public identifier is a publicly known name or value that can be used to to retrieve a file or data stream containing the contents of some entity.
`http://www.w3.org/TR/xml/#dt-pubid*` page(s) 6, 158

Qualified Name (QName)

A qualified name is either a normal name or a *prefixed name*. A name is either in no namespace or in the default namespace, whereas a *prefixed name* is always in some namespace.
`http://www.w3.org/TR/2006/REC-xml-names-20060816/#NT-QName`

`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#QName*` page(s) 10, 11, 45, 46, 48, 49, 58, 97, 145, 158

relative URI reference

A relative *URI reference* section 4.2 RFC3986[25] section 5 RFC2396[26]* page(s) 25, 27, 44, 89, 148, 153, 158

root node

The root node in *XPath* corresponds to the *document node* in *DOM* and to an *XML* file/stream.* page(s) 16–18, 140, 159

same-document reference

A *URI reference* that refers within the same-document. See section 2.3.4 and Figure 2.3.15.

- section 4.4 RFC3986[25]
- section 4.2 RFC2396[26]

* page(s) 17, 25–29, 41, 56, 59, 65, 66, 95, 113, 153, 158

Simple API for XML (SAX)

SAX is the Simple API for XML, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a de facto standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java.
`http://sax.sourceforge.net/*` page(s) 14–16, 44, 77, 96, 102, 105, 108, 110, 114–117, 149, 155, 158

Schema Centric XML Canonicalization Version 1.0 *OASIS* UDDI Spec TC 5 May 2005

An *Infoset* based canonicalization [56] that normalizes valid *XML* that has been assessed by *Schema*. See subsection 2.5.4.* page(s) 42, 47–49, 84–86, 143, 158

Standard Generalized Markup Language (SGML ISO 8879)

The Standard Generalized Markup Language is an ISO standard for representing text in electronic form, independent of device or system.
`http://www.w3.org/MarkUp/SGML/`,
`http://www.w3.org/TR/NOTE-sgml-xml-971215.html*` page(s) 4

simple inheritable attribute

Simple inheritable attributes, whose value that requires only a simple redeclaration, like *xml:lang* and *xml:space*.
`http://www.w3.org/TR/xml-c14n11/#dt-SimpleHeritableAtts`

`http://lists.w3.org/Archives/Public/public-xml-core-wg/2006Mar/0040.html`* page(s) 44, 58, 150, 158

simple type

A simple type in *Schema* provides a lexical or value space.
`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#rf-defn`* page(s) 12, 84, 137, 158, 159

SOAP

SOAP formerly was an acronym for Simple Object Access Protocol or Service Oriented Architecture Protocol but is today not using any of these acronyms any more. It is an XML protocol for decentralized, distributed environments.
`http://www.w3.org/TR/soap/`* page(s) 53, 80, 88, 137

Streaming API for XML (StAX)

`http://www.xml.com/pub/a/2003/09/17/stax.html`* page(s) 14–16, 77, 110

system identifier

The system identifier is converted to a *URI* to be dereferenced to retrieve a file or data stream.
`http://www.w3.org/TR/xml/#dt-sysid`* page(s) 6, 158

Technical Architecture Group (TAG)

The *TAG* is concerned with the Web architecture. `http://www.w3.org/2001/tag/`* page(s) 48, 147

Technical Committee (TC)

The *OASIS* term for working group (*WG*).* page(s) 1, 2, 47, 52, 53, 66, 74–76, 96–98, 118, 141, 148, 158

Transport Layer Security (TLS)

Transport Layer Security (TLS) `http://tools.ietf.org/html/rfc4346`* page(s) 53, 68

Transformation API for XML (TrAX)

The Transformation API for XML is an APIs for processing transformation instructions, and performing a transformation like XSLT.
`http://java.sun.com/webservices/jaxp/`* page(s) 32, 35

Uniform Resource Identifier (URI)

Is an Identifier of a certain syntax that is used as identify or resource identifier. See 2.3.4.
`http://tools.ietf.org/html/rfc2396`
`http://tools.ietf.org/html/rfc2732`
`http://tools.ietf.org/html/rfc3986*` page(s) 13, 15, 16, 23–32, 37, 39–41, 49, 52, 55, 58, 62, 65, 72, 73, 79, 80, 95, 103, 137, 147, 148, 153, 158

URI reference

A *URI* reference is either already a *URI* or it can be a *relative URI reference*. section 4.1 RFC3986[25] section 4 RFC2396[26]* page(s) 6, 25–28, 31, 41, 52, 56, 59, 60, 78, 89, 95, 146, 151, 158

visibly utilized namespace declaration

An Element visibly utilizes a namespace declaration if itself or one of its attributes uses the prefix declared by the namespace declaration. A namespace declaration is visibly utilized by its parent element if the parent or one of its attributes uses the prefix declared by the namespace declaration.
`http://www.w3.org/TR/xml-exc-c14n/#def-visibly-utilizes*` page(s) 44–46, 158

working group (WG)

The *W3C* term for *TC*.* page(s) 42, 147

Web Services Security (WSS)

Web Services Security (WSS) specifies set of SOAP extensions for securing web services by means of SAML, Kerberos, X.509, username tokens etc.
`http://www.oasis-open.org/committees/wss/*` page(s) 53, 75

OASIS Web Services Secure Exchange (WS-SX)

Web Services Secure Exchange OASIS WS-SX *TC* defines extensions to OASIS Web Services Security to enable trusted SOAP message exchanges.
`http://www.oasis-open.org/committees/ws-sx*` page(s) 53

World Wide Web Consortium (W3C)

The World Wide Web Consortium develops inter-operable technologies (specifications,guidelines, software, and tools) to lead the Web to its full potential. W3C is a forum for information, commerce, communication, and collective understanding.
`http://www.w3.org/*` page(s) 3, 6, 7, 9, 11, 22, 28, 30, 32, 35, 36, 49, 84, 148, 151

XML Advanced Electronic Signatures (XAdES)

XML Advanced Electronic Signature (XAdES) defines various properties for *XMLDSIG* signatures. These properties range from the signing time and signature policy over the signing certificate to properties needed for long term verification and archival of electronic signatures. [46]* page(s) 9, 13, 36, 53, 74, 82, 83, 89

Xalan

Apache Xalan is a project providing an *API* for evaluation of *XPath* expressions and an *XSLT* processor for transforming *XML* documents into *HTML*, text, or other *XML* document types.It is built on *SAX* 2 and *DOM* level 3.
`http://xml.apache.org/xalan-j/*` page(s) 95, 115, 127, 155, 158

Xerces

Apache Xerces is a project providing XML parsers, however in the context of this document, we are only concerned with the Xerces2 Java Parser
`http://xerces.apache.org/xerces2-j/*` page(s) 14, 16, 18, 115, 139, 158

XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)

XHTML is a reformulation of HTML 4 as an XML 1.0 application.
`http://www.w3.org/XML/*` page(s) 4–6, 32, 62, 99, 153

Extensible Markup Language (XML)

Extensible Markup Language is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.
`http://www.w3.org/XML/*` page(s) viii, 2–5, 7–19, 22, 27–29, 31, 32, 34, 35, 37–49, 51, 53, 54, 57–59, 61, 62, 64, 72, 74, 77–80, 82–90, 96–99, 102–104, 110–112, 114, 115, 120, 124, 139, 140, 144–146, 149–151, 154, 155, 158

xml declaration

The *XML* declaration is optional and the first element the *prolog*. It contains information about the version, optional information about the character encoding and an optional validity constraint able to assure a document is self-contained.
`http://www.w3.org/TR/xml/#NT-XMLDecl*` page(s) 6, 44, 79–81, 97, 99, 158

XML namespace

The namespace value `http://www.w3.org/XML/1998/namespace` is bound by definition to the `xml:` prefix.* page(s) 44, 149, 150, 158

xml:base

The attribute *xml:base* is an attribute defined in the *XML namespace*, that is used in *XML* documents to establish a *base URI reference* that is distinct from the surrounding file or parsed entity on the granularity of an element.[84]* page(s) 44–46, 58, 78, 80, 96, 138, 149, 150, 158

xml:id

The attribute *xml:id* is an `ID` attribute defined in the *XML namespace*, that is used in *XML* documents to associate the type `ID` with the attribute in the absence of validation.[24]* page(s) 15, 44, 138, 150, 158

xml:lang

Is a *simple inheritable attribute* and identifies the natural or formal language in which the content is written.
`http://www.w3.org/TR/REC-xml/#sec-lang-tag`[6]* page(s) 44, 46, 58, 78, 96, 146, 158

xml:space

The value default signals that applications' default white-space processing modes are acceptable for this element; the value preserve indicates the intent that applications preserve all the white space. `http://www.w3.org/TR/REC-xml/#sec-white-space`[6]* page(s) 44, 46, 58, 78, 82–84, 96, 99, 146, 154, 158

XML Core Working Group (XMLCORE)

Is chartered to consider comments on the following specifications: *XML*, *XML 1.1*, *XMLNS*, *XMLNS 1.1*, *Infoset*, *xml:base*, *xml:id*, *C14n*, *Exc-C14n*, and others. `http://www.w3.org/XML/Core/`* page(s) 7, 32, 42

XML-Signature Syntax and Processing (XMLDSIG)

XML-Signature Syntax and Processing specifies XML digital signature processing rules and syntax. XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.[29]* page(s) 2, 3, 9, 11, 13, 15, 17, 18, 22–25, 27–32, 35–39, 41, 43, 47, 52, 53, 55, 59–61, 63, 64, 66, 69, 72, 77, 78, 82–91, 93, 95, 99, 102, 106, 108, 110, 111, 113–115, 141, 143, 148, 150, 151, 153, 158

XML Signature Syntax and Processing - Second Edition - (XMLDSIG SE)

XML Signature Syntax and Processing - Second Edition - is a revision of *XMLDSIG*. It includes and addresses errata, differences between [RFC 4514] and [RFC 2253], recommends *C14n11* and recognizes the defect that *XPointer* is only a working draft. [29]* page(s) 23, 27–30, 32, 44

Extensible Markup Language 1.1

Extensible Markup Language 1.1 extended the character set for names in *XML 1.1* but since *XML* fifth edition this has been pulled into *XML* 1.0 as an erratum. Despite some additional control characters and their escaping the only real difference is that namespace prefix undeclarations are allowed in *XML 1.1*. `http://www.w3.org/TR/xml11/*` page(s) 7, 10, 11, 90, 94, 150, 154

Namespaces in XML 1.0 (Second Edition)

Namespaces allow to use multiple markup vocabularies by associating them with a namespace hence avoiding name collisions and allowing their recognition. `http://www.w3.org/TR/REC-xml-names/*` page(s) 9–11, 43, 48, 49, 150, 159

Namespaces in XML 1.1 (Second Edition)

Namespaces in XML 1.1 allow to undeclare a namespace binding. `http://www.w3.org/TR/xml-names11/*` page(s) 10, 11, 90, 150

XML Schema (XMLSchema)

The purpose of a schema is to define a class of XML documents, and so the term instance document is often used to describe an *XML* document that conforms to a particular schema. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/#Intro*` page(s) viii, 2, 5–9, 11–14, 16, 23, 32, 36, 42–44, 47–49, 51, 56, 57, 68, 71, 73–76, 80, 84–86, 88, 102, 103, 111–113, 122, 123, 137, 139, 145–147, 153, 155, 158, 159

XML Path Language (XPath) Version 1.0

XPath is primarily used to address parts of an XML document and manipulates strings, numbers and booleans. [32]* page(s) 4, 7, 11, 14, 15, 17–22, 28–30, 32–34, 38, 39, 44–46, 48, 49, 73, 78, 82, 86, 87, 90, 92–96, 113, 114, 138, 140, 141, 146, 149, 153, 154, 159

XML Pointer Language (XPointer) Version 1.0

XPointer is used to identify fragments by means of a *URI reference* locating a resource that is an XML document. XPointer until today never became a *W3C* recommendation and has the status of a candidate recommendation. Hence it shouldn't be normatively referenced. `http://www.w3.org/TR/2001/CR-xptr-20010911/` It has been superseded by the XPointer Framework. `http://www.w3.org/TR/xptr-framework/*` page(s) 16, 18, 27–31, 40, 41, 48, 73, 82, 92, 93, 95, 150, 155, 159

XML Security Toolkit (XSECT)

IAIK XML Security Toolkit (XSECT) implements the *JSR105* API for *XMLDSIG* in Java™ and JSR106 APIs for XML Encryption. `http://jce.iaik.tugraz.at/sic/products/xml_security/xsect/*` page(s) 15, 30, 60, 102, 103, 110

Extensible Stylesheet Language (XSL)

Extensible Stylesheet Language (XSL), a language for expressing stylesheets. It consists of two parts: 1. a language for transforming XML documents, and 2. an XML vocabulary for specifying formatting semantics. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.
`http://www.w3.org/TR/xsl/*` page(s) 5, 32, 33

Extensible Stylesheet Language (XSL-FO)

see XSL* page(s) 32

Extensible Stylesheet Language for Transformation (XSLT)

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.
`http://www.w3.org/TR/xslt/*` page(s) 5, 15, 18, 32–35, 73, 82, 87, 89, 92, 95, 96, 99, 114, 121, 149, 155, 159

XML Security Specifications Maintenance Working Group (XSSMWG)

The mission of this Working Group is to perform limited maintenance work on the basic XML Security specifications, and suggest a charter for further work.
`http://www.w3.org/2007/xmlsec/*` page(s) 30, 36

List of Figures

Index