Internet Draft
Document: draft-orri-spki-xml-cert-struc-00.txt
Category: Informational
Expires: May 2002

X. Orri
J.M. Mas
Octalis SA

November 2001

SPKI-XML Certificate Structure

<draft-orri-spki-xml-cert-struc-00.txt>

Status of this Document

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
http://www.ietf.org/ietf/1id-abstracts.txt
The list of Internet-Draft Shadow Directories can be accessed at
http://www.ietf.org/shadow.html.

Distribution of this document is unlimited. Comments should be sent to the authors (mas@octalis.com and orri@octalis.com).

Abstract

This draft suggests a standard form for transforming SPKI certificates encoded using S-expressions from and to XML documents. We present a XML Schema for the encoding and validation of SPKI certificates and other SPKI objects such as sequences and ACLs, and discuss different possibilities for the transformation of S-expressions into an XML document and vice-versa. The XML Schema is based on the "SPKI Certificate Structure" [SPKI].

The main emphasis of this document is on the encoding of all SPKI constructs under XML. Additionally, this draft provides a short discussion on specific possibilities for the transformation of S-expression encoded certificates to and from XML encoded certificates. The SPKI Certificate Theory is explained in [RFC2693]; it is not the intention or the objective of this document to address certificate design issues.

Table of Contents

1 Overview of Contents

This document represents a continuation to some, a different approach to others, of the work initiated by J. Paajarvi relative to the XML encoding of SPKI certificates in [PAAJ]. The authors feel both initiatives share the same goal, but take different approaches. The work in this document is based on XML Schemas instead of DTDs. [PAAJ] defines a DTD that somewhat "breaks" the syntax as defined in [SPKI] and make the trans-coding from/to XML to/from S-expressions rather complex. In the present document this trans-coding was one of the design goals. Furthermore, [PAAJ] is based on XML digital signatures as defined in [XSIG]. The authors do not believe this is the best approach in this case.

Our main objective when specifying the XML Schema has been to follow as much as possible the syntax and semantics defined by SPKI in [SPKI]. In some cases, if thinking exclusively in XML, one can easily find a simpler and better way to express a given part. Our main goal was not that of defining an XML Schema for certification, but rather defining an XML Schema for the XML encoding of SPKI certificates such that trans-coding from and to S-expressions is simple, and using standard tools whenever possible. We provide the corresponding DTD to the above-mentioned XML Schema for people who uses them.

The first sections of this document and its structure match that of the SPKI Certificate Structure [SPKI] as much as possible. Our intention is to facilitate the reading of this document to those already familiarized with the specification of SPKI certificates.

This document contains the following sections:

Section 1: this overview.

Section 2: a glossary of terms and definitions specific to SPKI and XML.

Section 3: the definition of SPKI structure primitives in XML used throughout this document.

Section 4: the definition of authorization certificates and its component parts.

Section 5: the definition of name certificates and those few parts that differ from authorization certificates.

Section 6: the definition of ACLs and sequence structures.

Section 7: the definition of online test reply formats.

Section 8: we discuss a possible solution for the trans-coding of S-expressions based SPKI certificates onto XML-encoded ones.

Section 9: we present and discuss some options for the trans-coding from XML-based onto S-expressions based SPKI certificates. We describe more in detail a XSL Stylesheet used for this process.

Section 10: we discuss open issues regarding the XML-SPKI encoding and compatibility issues.

Appendix A: we provide some examples of different SPKI constructs encoded in XML according to the XML Schema presented in this document. These range from simple examples extracted and adapted from

Appendix B: the full XML Schema for SPKI.

Appendix C: the full XML Schema for S-expressions.

Appendix D: the full XSL Stylesheet for the transformation of XML-encoded certificates onto S-expressions.

Appendix E: the DTD derived from the XML Schema.

The References section lists all documents and sources of relevant information referred to in the text, as well as readings which may be of interest to anyone reading on this topic.

The Acknowledgements section.

The Author's Addresses section gives the addresses, telephone numbers and e-mail addresses of the authors.

2 Glossary of Terms

In this section we define the terms used in the document. Most of the definitions regarding SPKI are taken directly from [SPKI] and [RFC2693], whereas those related to XML have been taken from various sources on the Internet.

2.1 SPKI Glossary

ACL: an Access Control List: a list of entries that anchors a certificate chain. Sometimes called a "list of root keys", the ACL is the source of empowerment for certificates. That is, a certificate communicates power from its issuer to its subject, but the ACL is the source that power (since it theoretically has the owner of the resource being controlled as its implicit issuer). An ACL entry has potentially the same content as a certificate body, but has no Issuer (and is not signed). There is most likely one ACL for each resource owner, if not for each controlled resource.

CERTIFICATE: a signed instrument that empowers the Subject. It contains at least an Issuer and a Subject. It can contain validity conditions, authorization and delegation information. Certificates come in three categories: ID (mapping <name,key>), Attribute (mapping <authorization,name>), and Authorization (mapping <authorization,key>). An SPKI authorization or attribute certificate can pass along all the empowerment it has received from the Issuer or it can pass along only a portion of that empowerment.

CANONICAL S-EXPRESSION: an encoding of an S-expression that does not permit equivalent representations and is designed for easy parsing.

FULLY QUALIFIED NAME: a local name together with a global identifier defining the name space in which that local name is defined.

GLOBAL IDENTIFIER: a globally unique byte string, associated with the keyholder. In SPKI this is the public key itself, a collision-free hash of the public key or a Fully Qualified Name.

HASH: a cryptographically strong hash function, assumed to be collision resistant. In general, the hash of an object can be used wherever the object can appear. The hash serves as a name for the object from which it was computed.

ISSUER: the signer of a certificate and the source of empowerment that the certificate is communicating to the Subject.

KEYHOLDER: the person or other entity that owns and controls a given private key. This entity is said to be the keyholder of the keypair or just the public key, but control of the private key is assumed in all cases.

NAME: a SDSI name always relative to the definer of some name space. This is sometimes also referred to as a local name. A global (fully qualified) name includes the global identifier of the definer of the name space. For example, if
(name jim)
is a local name,
(name (hash md5 |+gbUgUltGysNgewRwu/3hQ==|) jim)
could be the corresponding fully qualified name.

ONLINE TEST: one of three forms of validity test: (1) CRL; (2) revalidation; or (3) one-time revalidation. Each refines the date range during which a given certificate or ACL entry is considered valid, although the last defines a validity interval of effectively zero length.

PRINCIPAL: a cryptographic key, capable of generating a digital signature. We deal with public-key signatures in this document but any digital signature method should apply.

PROVER: the entity that wishes access or that digitally signs a document. The Prover typically sends a message or opens a channel to the Verifier that then checks signatures and credentials sent by the Prover.

SPEAKING: A Principal is said to "speak" by means of a digital signature. The statement made is the signed object (often a certificate). The Principal is said to "speak for" the Keyholder.

SUBJECT: the thing empowered by a certificate or ACL entry. This can be in the form of a key, a name (with the understanding that the name is mapped by certificate to some key or other object), a hash of some object, or a set of keys arranged in a threshold function.

S-EXPRESSION: the data format chosen for SPKI/SDSI. This is a LISP-like parenthesized expression with the limitations that empty lists are not allowed and the first element in any S-expression must be a string, called the "type" of the expression.

THRESHOLD SUBJECT: a Subject for an ACL entry or certificate that specifies K of N other Subjects. Conceptually, the power being transmitted to the Subject by the ACL entry or certificate is transmitted in (1/K) amount to each listed subordinate Subject. K of those subordinate Subjects must agree (by delegating their shares along to the same object or key) for that power to be passed along. This mechanism introduces fault tolerance and is especially useful in an ACL entry, providing fault tolerance for "root keys".

VALIDITY CONDITIONS: a date range that must include the current date and time and/or a set of online tests that must succeed before a certificate is to be considered valid.

VERIFIER: the entity that processes requests from a prover, including certificates. The verifier uses its own ACL entries plus certificates provided by the prover to perform "5-tuple reduction", to arrive at a 5-tuple it believes about the prover: <self,prover,D,A,V>.

2.2 XML Glossary

ATTRIBUTE: extra information pertaining to an element that is stored in the start tag of an element (as name="value" pairs) or assigned default values in attribute declarations. Attributes and child elements may sometimes be interchanged; typically, attributes contain information about a particular element, not information that might stand on its own as an extra element.

DTD: Document Type Definition, a formal description of the structure of a document that may also provide some content information. DTDs effectively describe XML file formats, providing the vocabulary and allowable structure of the elements in an XML document. The DTD for a document is the combination of the internal and external subsets described by the document type declaration.

ELEMENT: each XML document contains one or more elements. They consist of a start tag, and end tag, and the information between the tags, which is often referred to as the contents. They are described by a DTD or schema, which provide a description of the structure of the data. Each element has a type, identified by name, and may have a set of attributes, and each attribute has a name and a value.

SGML: Standard Generalized Markup Language, often referred to as "the mother of all markup languages". The international standard for defining descriptions of structure and content of electronic documents (ISO 8879). XML is a subset of SGML designed to deliver SGML type information over the Web.

VALID XML DOCUMENT: XML that conforms to the rules defined in the XML specification, as well as the rules defined in the DTD or schema. The parser must understand the validity constraints of the XML specification and check the document for possible violations. If the parser finds any errors, it must report them to the XML application. The parser must also read the DTD, validate the document against it, and again report any violations to the XML application. Because all of this parsing and checking can take time and because validation might not always be necessary, XML supports the notion of the well-formed document.

WELL-FORMED XML DOCUMENT: XML that follows the XML tag rules listed in the W3C Recommendation for XML 1.0, but doesn't have a DTD or schema. A well-formed XML document contains one or more elements; it has a single document element, with any other elements properly nested under it; and each of the parsed entities referenced directly

or indirectly within the document is well formed. Well-formed XML documents are easy to create because they don't require the additional work of creating a DTD. Well-formed XML can save download time because the client does not need to download the DTD, and it can save processing time because the XML parser doesn't need to process the DTD.

XML: eXtensible Markup Language, a system for defining specialized markup languages that are used to transmit formatted data. XML is conceptually related to HTML, but XML is not itself a markup language. Rather it's a metalanguage, a language used to create other specialized languages.

XML DOCUMENT: A document object that is well formed, according to the XML recommendation, and that might (or might not) be valid. The XML document has a logical structure (composed of declarations, elements, comments, character references, and processing instructions) and a physical structure (composed of entities, starting with the root, or document entity).

XSD: Xml Schema Definition, a formal specification of element names that indicates which elements are allowed in an XML document, and in what combinations. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It defines whether an element is empty or can include text. The schema can also define default values for attributes. A schema is functionally equivalent to a DTD, but is written in XML. A schema also provides for extended functionality such as data typing, inheritance, and presentation rules. Consequently, the new schema languages are far more powerful than DTDs.

XSL: eXtensible Stylesheet Language, a language for expressing stylesheets. It consists of a language for transforming XML documents, and 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.

XSLT: eXtensible Stylesheet Language Transformation, a language for transforming XML documents into other XML documents. 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 style for an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

3 Primitives

SPKI uses S-expressions in canonical form as the format for SPKI objects. An S-expression is a list enclosed in matching "(" and ")". We assume the S-expression technology of [SEXP] with the restrictions that no empty lists are allowed and that each list must have a byte string as its first element. That first element is the "type" or "name" of the object represented by the list and must be a byte string.

3.1 S-expressions

There are some parts in the specification of SPKI that contain undefined data, but expressed in terms of S-expressions. These are, for example, the <general-op> or the <online-test> constructs as defined in [SPKI].

For this reason, even if we use XML for the encoding of SPKI objects, we provide here an XML Schema for S-expressions, our goal being not breaking the semantics of SPKI.

For a more detailed description on the use of S-expression in SPKI refer to [SPKI] document, section 3.

The BNF definition of canonical S-expressions [SEXP] is as follows:

```
<s-expr>:: "(" <byte-string> <s-part>* ")" ;
<s-part>:: <byte-string> | <s-expr> ;
<byte-string>:: <bytes> | <display-type> <bytes> ;
<bytes>:: <decimal> ":" {binary byte string of that length} ;
<decimal>:: <nzddigit> <ddigit>* | "0" ;
<nzddigit>:: "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
<ddigit>:: "0" | <nzddigit> ;
<display-type>:: "[" <bytes> "]" ;
```

3.1.1 <bytes>

The <bytes> construct represents a sequence of binary bytes (octets). It is important to note that any content of an XML document must be printable. The term printable is in concordance with the definition in [XML] section 2.2. Basically, the representation of <bytes> in an XML document will correspond to the advanced form of S-expressions. The definition hereafter specifies the patterns for canonical (iff al bytes are XML-printable), base 64, hexadecimal, token and advanced text.

```
<xsd:simpleType name="bytes" id="bytes">
 <xsd:restriction base="xsd:binary">
  <xsd:pattern value="(\p{Nd})+:(\p{L}|\p{M}|\p{N}|
                      \p{P}|\p{Z}|\p{S}|\p{C})+"/>
  <xsd:pattern value="\|(.)+\|"/>
  <xsd:pattern value="#([0-9]|[A-F]|[a-f])*#"/>
  <xsd:pattern value='"(.)*"'/>
  <xsd:pattern value="([a-zA-Z\-\./_:\*\+=]
                      [a-zA-Z0-9\-\./_:\*\+=]*)"/>
 </xsd:restriction>
</xsd:simpleType>
```

The first pattern defines a <bytes> construct in canonical form. Note that this pattern defines the length of the <bytes>, the ':' and the characters accepted, but cannot impose that the length specified corresponds with the number of bytes after. And remember that only those <bytes> whose all bytes are printable can be found in an XML document.

The second pattern defines a <bytes> construct in base-64 form. Here we have included the mark '|' as defined in [SPKI] in the XML Schema, but have not defined all legal characters in base-64. Note that the use of these patterns does not preclude the use of attributes specifying the type of encoding. This can be added if necessary.

The third pattern defines a <bytes> construct in hexadecimal form. Here too, we have included the marks '#' that SPKI defines for hexadecimal representation of <bytes>.

The fourth pattern defines a <bytes> construct in advanced text. We have also included here the mark '"' as defined in SPKI for strings encoded in advanced form. Here we accept any XML printable character as valid. IMPORTANT: we have found an incompatibility between what SPKI defined as printable characters and what XML currently defines as printable characters: the vertical tab. The W3C is currently addressing these special characters and we foresee the issue solved in future specifications of XML.

The last pattern defines a <bytes> construct that corresponds to a token. A token is a sequence of bytes starting by a letter or simple punctuations followed by any letter, digit or simple punctuation.

It is important to note that the use of marking characters in the representation of bytes has as objectives, on the one hand the respect of the syntax defined by SPKI, and on the other hand, make the process of trans-coding from XML back to S-expressions straightforward using XSL Stylesheets.

3.1.2 <byte-string>

A <byte-string> is a sequence of bytes that can optionally be modified by a display type. This display type is assumed to be a MIME type giving optional instructions to any program whishing to use the byte string. Byte-strings and display types are composed of <bytes> objects. The XML Schema definition of a <byte-string> can be found hereafter.

```
<xsd:complexType name="byte-string">
 <xsd:simpleContent>
  <xsd:extension base="bytes">
   <xsd:attribute name="display-type"
                  type="display-type" use="optional" />
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>

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

In the previous definition, the <byte-string> type is defined as extending a bytes type with the optional attribute <display-type>. Then we define the element <byte-string> as being of type <byte-string>. And the definition of display type is as follows:

```
<xsd:simpleType name="display-type">
 <xsd:restriction base="xsd:binary">
  <xsd:whiteSpace value="collapse"/>
  <xsd:pattern value="\[(\p{Nd})+:(\p{L}|\p{M}|\p{N}|
                      \p{P}|\p{Z}|\p{S}|\p{C})+\]"/>
  <xsd:pattern value="\[\|(.)+\|\]"/>
  <xsd:pattern value="\[#([0-9]|[A-F]|[a-f])*#\]"/>
  <xsd:pattern value='\["(.)*"\]'/>
  <xsd:pattern value="\[([a-zA-Z\-\./_:\*\+=]
                      [a-zA-Z0-9\-\./_:\*\+=]*)\]"/>
</xsd:restriction>
</xsd:simpleType>
```

3.1.3 <integer>

[SPKI] in section 3.2.1 claims that "An integer is a kind of byte-string, that we distinguish only because it is encoded in the way expected by multi-precision libraries." For this exact reason we felt that it should not be included in the S-expressions XML Schema but rather in the SPKI one. An integer does not change the syntax of an S-expression, but rather changes the semantics of a <byte-string>.

The following defines the <integer> type as simply being an extension of <byte-string>.

```
<xsd:complexType name="integer" id="integer">
 <xsd:simpleContent>
  <xsd:extension base="byte-string"/>
 </xsd:simpleContent>
</xsd:complexType>
```

3.1.4 <sexpr> and <spart>

An S-expression, by definition, is a list with a type and zero or more parts. These parts are called S-parts and can be whether another S-expression or a byte-string.

The XML Schema definition for S-expressions and S-parts is as follows:

```
<xsd:group name="spart">
 <xsd:choice>
  <xsd:element ref="byte-string"/>
  <xsd:element ref="sexpr"/>
 </xsd:choice>
</xsd:group>

<xsd:complexType name="sexpr">
 <xsd:sequence>
  <xsd:element name="type" type="byte-string" id="type-sexpr"/>
  <xsd:group ref="spart" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

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

The definition of <spart> states that an S-part type is whether a <sexpr> or a <byte-string>, whereas an S-expression is defined as having a type or name <byte-string> plus any number of <spart>'s. Finally we define the element <sexpr> ad being of type <sexpr>.

3.2 Primitive Objects

SPKI builds on a set of primitive objects, those directly related to principals. These primitive objects are public keys that represent a principal (speak for), the corresponding private keys that issue signatures, the hashes of public keys as an equivalent to public keys, and signatures for the authenticity of statements.

In this section we provide the XML Schema definition of these primitive objects and explain them in some detail.

3.2.1 <public-key>

A public key definition gives everything the user needs to employ the key for checking signatures. Until today, SPKI has defined two algorithms for signature keys: RSA and DSA. Elliptic Curve Cryptography algorithm identifiers (and keys) will most likely be defined too by SPKI.

```
<xsd:element name="public-key" type="pub-key"/>

<xsd:complexType name="pub-key">
 <xsd:sequence>
  <xsd:element ref="key-value"/>
  <xsd:element ref="uris" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

We define a <public-key> element as being of type <pub-key>. Following the above definition, the <pub-key> type is defined as having a key value and, optionally, a <uris> object containing URI's to certificates empowering the public key.

3.2.1.1 <key-value> and <pub-sig-alg-id>

The standardized algorithms are defined hereafter. If at a later time SPKI changes the list of standardized algorithm identifiers, we should add them to this type, to the key value type along with the definition of the corresponding key values.

```
<xsd:element name="pub-sig-alg-id" type="pub-sig-alg-id"/>

<xsd:simpleType name="std-pub-sig-alg-id">
 <xsd:restriction base="xsd:string">
  <xsd:whiteSpace value="collapse"/>
  <xsd:enumeration value="13:rsa-pkcs1-md5"/>
  <xsd:enumeration value="rsa-pkcs1-md5"/>
  <xsd:enumeration value="14:rsa-pkcs1-sha1"/>
  <xsd:enumeration value="rsa-pkcs1-sha1"/>
  <xsd:enumeration value="9:rsa-pkcs1"/>
  <xsd:enumeration value="rsa-pkcs1"/>
  <xsd:enumeration value="8:dsa-sha1"/>
  <xsd:enumeration value="dsa-sha1"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="pub-sig-alg-id">
 <xsd:union memberTypes="std-pub-sig-alg-id xsd:string"/>
</xsd:simpleType>
```

The above defines the type <pub-sig-alg-id> as whether a standard algorithm identifier or any other identifier as a string value. We assume here that we cannot have algorithms identifiers with display types (see section 10 for a discussion on this issue). The structure of the document is the same for standard and non-standard algorithms. Strictly speaking, the definition of standard identifiers is not necessary. We chose to define it for concordance with the SPKI specification [SPKI]. We address hashing algorithm identifiers in the same way.

Note that the actual values for algorithm identifiers defined correspond both to canonical and advanced representation of byte-strings. Doing so we minimize inherent trans-coding problems that might appear.

```
<xsd:element name="key-value" type="key-value"/>

<xsd:complexType name="key-value">
 <xsd:sequence>
  <xsd:element ref="pub-sig-alg-id"/>
  <xsd:choice>
   <xsd:element ref="rsa-key-value"/>
      <xsd:element ref="dsa-key-value"/>
   <xsd:element ref="any-key-value"/>
  </xsd:choice>
 </xsd:sequence>
</xsd:complexType>
```

Like with algorithm identifiers, if SPKI standardizes a new algorithm, we will need to add it/them to this definition and provide the corresponding key values.

From a XML point of view, we should allow for key values defined in "any" namespace. This way the concept of key value becomes extensible by simply defining the new key values in another, probably custom, namespace. We could still validate the XML document. But the case of SPKI we cannot allow it for one reason: we want to be able to transform those key values back to S-expressions. And in order to do so, the stylesheet needs to "know" the XML document in order to transform them to S-expressions. The same restriction applies to private key values, hashes and signatures. We will see this issue more in detail in section 10.

3.2.1.2 RSA Public Key Value

```
<xsd:element name="rsa-key-value" type="rsa-key-value"/>

<xsd:complexType name="rsa-key-value">
 <xsd:sequence>
  <xsd:element name="n" type="byte-string"/>
  <xsd:element name="e" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>
```

3.2.1.3 Example of XML-encoded RSA Public Key

```
<public-key>
 <key-value>
  <pub-sig-alg-id>rsa-pkcs1-md5</pub-sig-alg-id>
  <rsa-key-value>
   <n>|ANHCG85jXFGmicr3MGPj53FYYSY1aWAue6PKnpFErHhKMJa4HrK4WSKTOYTTl
      apRznnELD2D7lWd3Q8PD0lyi1NJpNzMkxQVHrrAnIQoczeOZuiz/yYVDzJ1Ddi
      Imixyb/Jyme3D0UiUXhd6VGAz0x0cgrKefKnmjy410Kro3uW1|</n>
   <e>#03#</e>
  </rsa-key-value>
 </key-value>
</public-key>
```

3.2.1.4 DSA Public Key Value

```
<xsd:element name="dsa-key-value" type="dsa-key-value"/>

<xsd:complexType name="dsa-key-value">
 <xsd:sequence>
  <xsd:element name="p" type="byte-string"/>
  <xsd:element name="g" type="byte-string" minOccurs="0"/>
  <xsd:element name="q" type="byte-string"/>
  <xsd:element name="y" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>
```

3.2.1.5 Example of XML-encoded DSA Public Key

```
<public-key>
 <key-value>
  <pub-sig-alg-id>dsa-sha1</pub-sig-alg-id>
  <dsa-key-value>
   <p>|AMxZt4PXzxBFGaF5r+cGpXQzNXCHjjk1awgnr4LCzXYbC97QVXi/Xes1k28t0
      YcDlon56Yut0lTz39fziBpHbGBfc1LvOgW1P5MIa1W8eM3UXi4dzWjWtjCn/QM
      2s33qyELDsCmgAeKg3sVygjKavNgZiSxf44R7RcIEnZBxkcN/|</p>
   <q>|AP9n7Cy++blLMxOaB0ML3Z3Cc+qh|</q>
   <g>|fbT/lMbMgBWb81X2kRyklLLO/TamsDbLCyp2esdrf/3771RKgsI1RZTWMxIpR
```

```
        51D6maNNpEywxhy4L8isXFXplysrAMCfDjpaUCowhQNSDRT8YzygxZHJpZIU8i
        t+QtLc4fIxA/qSqFL4N3fTIe7xApQlmmG9bI2lgBlZbi1/OU=|</g>
      <y>|ALpgrX32c8zRlqBSBMtvJzYwrXXpCj3oqeevPna/9zND2LX7wVZd1c9K6ZxmQ
        CqxDqGl/anDVToNAnlzr2btlS32cymsxpEm8bIlAJ6Jk4clT3NrxuTDRft/W+r
        gvndiK8fEmtNZ2iaYgAKoM2M3zbij6Ts1H0FfjODHZrtULyNB|</y>
     </dsa-key-value>
    </key-value>
   </public-key>
```

3.2.1.6 Non-standard Public Key Value

```
<xsd:complexType name="any-key-value">
 <xsd:sequence>
  <xsd:element ref="sexpr" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

As already pointed out in section 3.2.1.1, we cannot allow external namespaces for the definition of new key values. For this reason, non-standardized key values must conform to this specification. It will be at an application level that these key values should be dealt with.

3.2.2 <private-key>

Although it is not needed by the SPKI standard or by the XML encoding of SPKI, we provide the definition of the corresponding private keys for the public keys above.

```
<xsd:element name="private-key" type="priv-key"/>

<xsd:complexType name="priv-key">
 <xsd:sequence>
  <xsd:element ref="pub-sig-alg-id"/>
  <xsd:choice>
   <xsd:element ref="rsa-priv-key-value"/>
   <xsd:element ref="rsa-crt-key-value"/>
   <xsd:element ref="dsa-priv-key-value"/>
   <xsd:element ref="any-key-value"/>
  </xsd:choice>
 </xsd:sequence>
</xsd:complexType>
```

Note that the definition of a private key does not rely on a private key value (like with public keys). This is because private keys do not require a <uris> object attached to them, so basically a private key is a key value and nothing more. We see hereafter the definition of private keys for the algorithms defined in [SPKI].

3.2.2.1 RSA and CRT Private Key Values

In the definitions hereafter, note that the e parameter is optional. It is current practice in today's software implementations to provide the public coefficients within private keys. For this reason we allow, optionally, the inclusion of the e coefficient here. It will be the case too for DSA private keys.

```
<xsd:complexType name="rsa-priv-key-value">
 <xsd:sequence>
  <xsd:element name="e" type="byte-string" minOccurs="0"/>
  <xsd:element name="n" type="byte-string"/>
  <xsd:element name="d" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="rsa-crt-key-value">
 <xsd:sequence>
  <xsd:element name="e" type="byte-string" minOccurs="0"/>
  <xsd:element name="n" type="byte-string"/>
  <xsd:element name="d" type="byte-string"/>
  <xsd:element name="p" type="byte-string"/>
  <xsd:element name="q" type="byte-string"/>
  <xsd:element name="a" type="byte-string"/>
  <xsd:element name="b" type="byte-string"/>
  <xsd:element name="c" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>
```

3.2.2.2 Example of XML-encoded RSA Private Key

```
<private-key>
 <pub-sig-alg-id>rsa-pkcs1-md5</pub-sig-alg-id>
  <rsa-crt-key-value>
  <e>#03#</e>
  <n>|ANHCG85jXFGmicr3MGPj53FYYSY1aWAue6PKnpFErHhKMJa4HrK4WSKTOYTTla
     pRznnELD2D7lWd3Q8PD0lyi1NJpNzMkxQVHrrAnIQoczeOZuiz/yYVDzJ1DdiIm
     ixyb/Jyme3D0UiUXhd6VGAz0x0cgrKefKnmjy410Kro3uW1|</n>
  <d>|AIvWvTRCPYvEW9ykyu1CmkuQQMQjm5V0Um0xvwuDHaWGyw8lacx65hcM0QM3uR
     w2iaaCyCkCnuO+k19fX4ZMXOD7cLN/Qrql8Efx5mczcoGN+Eo6FF+cvgXfupe1V
     M6PmJdFIauJerTHUOlPrI12N+NnAL7CvU6X1nhOnf/Z77iz|</d>
  <p>|APesjZ8gK4RGV5Qs1eCRAVp7mVblgf13R5fwApw6bTVWzunIwk/2sShyytpc90
     edr+0DPwldnvEXTUY1df0DwPc=|</p>
  <q>|ANjPQe6O0Jfv90GWE3q2c9724AX7FKx64g2F8lxgiWW0QKEeqiWiiEDx7qh0lL
     rhmBT+VXEDFRG2LHmuNSTzj7M=|</q>
  <a>|AKUds79qx62EOmLIjpW2AOb9EOSZAVOk2mVKrGgm83jkifEwgYqkdhr3MebopN
     ppH/NXf1uTv0tk3i7OTqitK08=|</a>
  <b>|AJCKK/RfNbqf+iu5YlHO9+n56q6nYx2nQV5ZTD2VsO54KxYUcW5sWtX2nxr4Yy
     dBEA3+46CsuLZ5cvvJeMNNCnc=|</b>
  <c>|CIPwAAO8Vmj0/BfCtsg+35+r94jwxGYHZ63RsqyNxbvkAO6xPqSht8/vzdR93e
     X5B9ZKBQg1HHWCsHbqQtmNLQ==|</c>
```

```
    </rsa-crt-key-value>
   </private-key>
```

3.2.2.3 DSA Private Key Value

```
   <xsd:complexType name="dsa-priv-key-value">
    <xsd:sequence>
     <xsd:element name="p" type="byte-string"/>
     <xsd:element name="g" type="byte-string" minOccurs="0"/>
     <xsd:element name="q" type="byte-string"/>
     <xsd:element name="y" type="byte-string"/>
     <xsd:element name="x" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>
```

3.2.2.4 Example of a XML-encoded DSA Private Key

```
   <private-key>
    <pub-sig-alg-id>dsa-sha1</pub-sig-alg-id>
    <dsa-priv-key-value>
     <p>|AMxZt4PXzxBFGaF5r+cGpXQzNXCHjjk1awgnr4LCzXYbC97QVXi/Xes1k28t0Y
        cDlon56Yut0lTz39fziBpHbGBfc1LvOgW1P5MIa1W8eM3UXi4dzWjWtjCn/QM2s
        33qyELDsCmgAeKg3sVygjKavNgZiSxf44R7RcIEnZBxkcN/|</p>
     <g>|fbT/lMbMgBWb81X2kRyklLLO/TamsDbLCyp2esdrf/3771RKgsI1RZTWMxIpR5
        1D6maNNpEywxhy4L8isXFXplysrAMCfDjpaUCowhQNSDRT8YzygxZHJpZIU8it+
        QtLc4fIxA/qSqFL4N3fTIe7xApQlmmG9bI2lgBlZbi1/OU=|</g>
     <q>|AP9n7Cy++blLMxOaB0ML3Z3Cc+qh|</q>
     <y>|ALpgrX32c8zRlqBSBMtvJzYwrXXpCj3oqeevPna/9zND2LX7wVZd1c9K6ZxmQC
        qxDqGl/anDVToNAnlzr2btlS32cymsxpEm8bIlAJ6Jk4clT3NrxuTDRft/W+rgv
        ndiK8fEmtNZ2iaYgAKoM2M3zbij6Ts1H0FfjODHZrtULyNB|</y>
     <x>|ALpglLLO/TamsDbLCyp2tvJzYwrXXpCj3oqeevPna/9zND2LX7wVZd1c9K6Zxm
        QCqxDqGl/anDVToNAnlzr2btlS32cymsxpEm8bIlAJ6Jk4clT3NrxuTDRft/W+r
        gvndiK8fEmtNZ2iaYgAKoM2M3zbij6Ts1H0FfjODHZrtULyNB|</x>
    </dsa-priv-key-value>
   </private-key>
```

3.2.2.5 Non-standard Private Key Value

There is no difference between public and private key values whose algorithms have not been standardized within SPKI. The definition is the same as in section 3.2.1.6

3.2.3 <hash>

The <hash> object provides the hash of some other object. The <hash> type is defined as a sequence of elements containing the hashing algorithm, its value and, optionally, a <uris> object in case it is the hash of a public key.

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

<xsd:complexType name="hash">
 <xsd:sequence>
  <xsd:element ref="hash-alg-name"/>
  <xsd:element ref="hash-value"/>
  <xsd:element ref="uris" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

For <hash> objects, like with keys, one can find the definition of the standard SPKI hashing algorithms and their values.

```
<xsd:simpleType name="std-hash-alg-name">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="3:md5"/>
  <xsd:enumeration value="md5"/>
  <xsd:enumeration value="4:sha1"/>
  <xsd:enumeration value="sha1"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="hash-alg-name">
 <xsd:union memberTypes="std-hash-alg-name xsd:string"/>
</xsd:simpleType>
```

The names of hashing algorithms are treated in the same way as public key signature algorithms. See section 3.2.1.1 for further explanations.

Finally, the definition hereafter states that the hash value element is of type <byte-string>.

```
<xsd:element name="hash-value" type="byte-string"/>
```

3.2.3.1 Example of XML-encoded SHA-1 Hash

```
<hash>
 <hash-alg-name>sha1</hash-alg-name>
 <hash-value>#1a6f6d621abd4476f16d0800fe4c32d06ff62e93#</hash-value>
</hash>
```

3.2.3.2 Example of XML-encoded MD5 Hash

```
<hash>
 <hash-alg-name>md5</hash-alg-name>
 <hash-value>#9710f155723bc5f4e0422ea53ff7c495#</hash-value>
</hash>
```

3.2.4 <signature>

A signature object is typically used for signing a certificate body and follows that certificate in a <sequence>. One can also sign objects other than certificate bodies, for example an access request or a file.

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

<xsd:complexType name="signature">
 <xsd:sequence>
  <xsd:element ref="hash"/>
  <xsd:group ref="principal"/>
  <xsd:element ref="signature-value"/>
 </xsd:sequence>
</xsd:complexType>
```

The <signature> type is a sequence of different elements, namely, the hash of the object signed, the signing principal and the signature value.

The definition of principal we use here is that of an anonymous type, and then reference it from inside object definitions; in most cases we do not need to specify an extra element of type principal, but rather simply verify it is whether a public key or a hash. Or, in other words, validate that the rules of the type principal hold.

```
<xsd:group name="principal">
 <xsd:choice>
  <xsd:element ref="public-key"/>
  <xsd:element ref="hash"/>
 </xsd:choice>
</xsd:group>
```

There is one exception to the previous, and that is the issuer object: we do need to specify there that element <issuer> is of type <principal>. In order to permit this, we define too the type for principal as follows:

```
<xsd:complexType name="principal">
 <xsd:group ref="principal"/>
</xsd:complexType>
```

3.2.4.1 <signature-value>

The type <signature-value> is a sequence of the algorithm identifier (see section 3.2.1) and the signature parameters, which are dependant on the algorithm used for signing. SPKI has fully specified the signatures for RSA and DSA-based signatures. We see the corresponding type definition hereafter. In case SPKI

standardizes new signature algorithms, we would need to define them as a choice in the <sig-params> type and provide the appropriate type definition.

```
<xsd:element name="signature-value" type="signature-value"/>

<xsd:complexType name="signature-value">
 <xsd:sequence>
  <xsd:element ref="pub-sig-alg-id"/>
  <xsd:group ref="sig-params"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:group name="sig-params">
 <xsd:choice>
  <xsd:element ref="dsa-sig-params"/>
  <xsd:element ref="rsa-sig-params"/>
  <xsd:element ref="any-sig-params"/>
 </xsd:choice>
</xsd:group>
```

3.2.4.2 RSA Signature Parameters

```
<xsd:element name="rsa-sig-params" type="byte-string"/>
```

A RSA signature is an array of bytes. This defines the element <rsa-sig-params> as having type <byte-string>.

3.2.4.3 Example of XML-encoded RSA Signature

```
<signature>
 <hash>
  <hash-alg-name>sha1</hash-alg-name>
  <hash-value>|UNGhcpNFWg5UhtoV2yxV6wPMJPA=|</hash-value>
 </hash>
 <public-key>
  <key-value>
   <pub-sig-alg-id>rsa-pkcs1-sha1</pub-sig-alg-id>
   <rsa-key-value>
    <e>#11#</e>
    <n>|AMC7wEqs+AjILPsUmS+R1PV9OihhqSTfmdLo9Y2hdj7+2f31qxXsMpxZedTx
       mcW9RKsf7dRjnRTxY51/MOIn0isY3DV3fMiaT8NUrjf+jEjF91V1HtCPn7+MH
       Tv/quWToc9/n4BhVDxH1IspFteoW0RHtZqOUfQcSQNswt7yrXFN|</n>
   </rsa-key-value>
  </key-value>
 </public-key>
 <signature-value>
  <pub-sig-alg-id>dsa-sha1</pub-sig-alg-id>
  <rsa-sig-params>|UN0g7krgm6Xq6vvws+oDZes9hy0pwDV9gVjuUV+uRC8Y7TDh1
                  JPfv2dhXBXqgERa3q99GHxgyjoDgfFgl/fAOplwz3vySmnATIn
```

```
                    rtCMxGdXgZlQ/SQ5xFXz3VlKQHgak0rK4xpZEbsR6YMggcGK7N
                    jZWTfNK0q8v4FSSD9UDkxk=|</rsa-sig-params>
    </signature-value>
   </signature>
```

3.2.4.4 DSA Signature Parameters

```
   <xsd:complexType name="dsa-sig-params">
    <xsd:sequence>
     <xsd:element name="r" type="byte-string"/>
     <xsd:element name="s" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>
```

A DSA signature is composed of two parts, the r and s coefficients. The above definition specifies this.

3.2.4.5 Example of XML-encoded DSA Signature

```
   <signature>
    <hash>
     <hash-alg-name>sha1</hash-alg-name>
     <hash-value>|UNGhcpNFWg5UhtoV2yxV6wPMJPA=|</hash-value>
    </hash>
    <public-key>
     <key-value>
      <pub-sig-alg-id>dsa-sha1</pub-sig-alg-id>
      <dsa-key-value>
       <p>|AMxZt4PXzxBFGaF5r+cGpXQzNXCHjjk1awgnr4LCzXYbC97QVXi/Xes1k28t
          0YcDlon56Yut0lTz39fziBpHbGBfc1LvOgW1P5MIa1W8eM3UXi4dzWjWtjCn/
          QM2s33qyELDsCmgAeKg3sVygjKavNgZiSxf44R7RcIEnZBxkcN/|</p>
       <g>|fbT/lMbMgBWb81X2kRyklLLO/TamsDbLCyp2esdrf/3771RKgsI1RZTWMxIp
          R51D6maNNpEywxhy4L8isXFXplysrAMCfDjpaUCowhQNSDRT8YzygxZHJpZIU
          8it+QtLc4fIxA/qSqFL4N3fTIe7xApQlmmG9bI2lgBlZbi1/OU=|</g>
       <q>|AP9n7Cy++blLMxOaB0ML3Z3Cc+qh|</q>
       <y>|ALpgrX32c8zRlqBSBMtvJzYwrXXpCj3oqeevPna/9zND2LX7wVZd1c9K6Zxm
          QCqxDqGl/anDVToNAnlzr2btlS32cymsxpEm8bIlAJ6Jk4clT3NrxuTDRft/W
          +rgvndiK8fEmtNZ2iaYgAKoM2M3zbij6Ts1H0FfjODHZrtULyNB|</y>
      </dsa-key-value>
     </key-value>
    </public-key>
    <signature-value>
     <pub-sig-alg-id>dsa-sha1</pub-sig-alg-id>
     <dsa-sig-params>
      <r>|APyNegTrlzLMCCcMRWoMlnKAOHIu|</r>
      <s>|AIPV/423068nuoNmoQQupyW3x+S1|</s>
     </dsa-sig-params>
    </signature-value>
   </signature>
```

3.2.4.6 Non-standard Signature Parameters

```
<xsd:complexType name="any-sig-params">
 <xsd:choice>
  <xsd:element ref="byte-string"/>
  <xsd:element ref="sexpr" maxOccurs="unbounded"/>
 </xsd:choice>
</xsd:complexType>
```

Like in [SPKI], we allow for signature schemas that have not been defined or standardized by SPKI. Doing so, application developers can use this XML Schema with the signature algorithm of their preference.

4 Authorization Certificates

The basic certificate form in SPKI is an authorization certificate. It transfers some specific authorization (or permission, right, credentials, etc.) from one principal to another. Authorization certificates do not deal with name definitions. These are addressed by name certificates and are explained in next section.

Because a certificate merely transfers authorizations, rather than creating them, we inject authorizations into a chain of through ACLs.

In order to deal with name and authorization certificates, and given that the differences between the two are few, we define an abstract certificate type in XML, and then extend it into name and authorization certificates.

The abstract certificate definition is given hereafter:

```
<xsd:element name="cert" type="certificate"/>

<xsd:complexType name="certificate">
 <xsd:sequence>
  <xsd:element ref="version" minOccurs="0"/>
  <xsd:element ref="display" minOccurs="0"/>
  <xsd:element ref="subject"/>
  <xsd:element ref="validity" minOccurs="0"/>
  <xsd:element ref="comment" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

And the definition of an authorization certificate is as follows:

```
<xsd:complexType name="authorization-cert">
 <xsd:complexContent>
  <xsd:restriction base="certificate">
   <xsd:sequence>
    <xsd:element ref="version" minOccurs="0"/>
    <xsd:element ref="display" minOccurs="0"/>
    <xsd:element ref="issuer"/>
    <xsd:element ref="issuer-info" minOccurs="0"/>
    <xsd:element ref="subject"/>
    <xsd:element ref="subject-info" minOccurs="0"/>
    <xsd:element ref="propagate" minOccurs="0"/>
    <xsd:element ref="tag"/>
    <xsd:element ref="validity" minOccurs="0"/>
    <xsd:element ref="comment" minOccurs="0"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>
```

The reader may wonder why do we need to redefine fields already defined in the abstract certificate back in the authorization certificate. The reason for this is to define the order in which the elements are to be found. According to [SPKI], certificates and ACLs are the only elements in SPKI whose parts are not positional; that is, their elements do not have any pre-determined order. But due to the strong limitations in XML for "set" support, we are unable to define types in XML whose elements are unordered. We see this more in detail in section 10.

4.1 <version>

```
<xsd:element name="version" type="integer"/>
```

The above definition imposes versions as integer numbers, and thus eliminates the possibility of versions under the form of major, minor, micro. This definition of version changed in the last release of [SPKI].

4.2 <display>

```
<xsd:element name="display" type="byte-string"/>
```

This optional field gives a display hint for the certificate. This field is ignored during the certificate chain reduction. Above we defined the element <display> as having type <byte-string>.

4.3 <issuer>

```
<xsd:element name="issuer" type="principal"/>
```

The issuer if of type principal, and the definition of principal (already given in previous section) is as follows:

```
<xsd:group name="principal">
 <xsd:choice>
  <xsd:element ref="public-key"/>
  <xsd:element ref="hash"/>
 </xsd:choice>
</xsd:group>

<xsd:complexType name="principal">
 <xsd:group ref="principal"/>
</xsd:complexType>
```

Note that this is the only case in which we will use the type principal. In all other cases, principal is an anonymous type.

4.4 <issuer-info>

```
<xsd:element name="issuer-info" type="uris"/>
```

The issuer info object provides the location of the certificate(s) by which the issuer derives the authority to pass along the authorization in the present certificate.

4.5 <subject>

```
<xsd:element name="subject" type="subj-obj"/>
```

The subject identifies the "thing" that is being empowered by this certificate. The <subj-obj> defines that "thing" as follows:

```
<xsd:group name="subject-obj">
 <xsd:choice>
  <xsd:group ref="principal"/>
  <xsd:element ref="name"/>
  <xsd:element ref="object-hash"/>
  <xsd:element ref="keyholder"/>
  <xsd:element ref="k-of-n"/>
 </xsd:choice>
</xsd:group>
```

The above definition states that a <subject-obj> can be whether a principal, a name, and object hash, a keyholder object or a k-of-n subject thresh.

Like with principals, here we need to define the type <subject-obj> as being an anonymous type for all cases except in the definition of subject, where a type is required (with the associated tag in the XML document). The type definition for <subj-obj> is as follows:

```
<xsd:complexType name="subj-obj">
 <xsd:group ref="subject-obj"/>
</xsd:complexType>
```

4.5.1 <object-hash>

```
<xsd:element name="object-hash" type="hash"/>
```

We define an <object-hash> element being of type <hash>. An object hash refers to a subject other than a principal (directly or through naming).

4.5.2 <keyholder>

```
<xsd:element name="keyholder" type="keyholder-obj"/>

<xsd:complexType name="keyholder-obj">
 <xsd:choice>
  <xsd:group ref="principal"/>
  <xsd:element ref="name"/>
 </xsd:choice>
</xsd:complexType>
```

We define the <keyholder> element as being of type <keyholder-obj>. And a keyholder object is in fact a choice between a principal (reference to the anonymous type) and a name.

A keyholder object refers to the flesh and blood (or iron and silicon) holder of a referenced key. A certificate with this subject tells us something about that person or machine.

4.5.3 <k-of-n>

```
<xsd:element name="k-of-n" type="subj-thresh"/>

<xsd:complexType name="subj-thresh">
 <xsd:sequence>
  <xsd:element ref="k-val"/>
  <xsd:element ref="n-val"/>
  <xsd:group ref="subject-obj" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

We define the element <k-of-n> as being of type <subj-thresh>, that, in turn, is defined as a sequence containing the values for k and n, and a list of subject objects.

4.6 <subject-info>

```
<xsd:element name="subject-info" type="uris"/>
```

The subject info object is nothing more than a <uris> object, providing the location information about the subject.

4.7 <propagate>

We define the element <propagate> having type <deleg>, which is an "empty" type. This allows having the tag <propagate/> without any contents in a XML document. Furthermore if the <propagate> tag has any contents, it will the treated as error.

```
<xsd:element name="propagate" type="deleg"/>

<xsd:complexType name="deleg"/>
```

4.8 <tag>

Tags in SPKI define the actual statement in a certificate. Tags can carry authorizations, credentials or keyholder location information.

The definition of the tag format in [SPKI] is, in our opinion, somewhat confusing, and can be found hereafter:

```
<tag>:: <tag-star> | "(" "tag" <tag-expr>  ")" ;
<tag-star>:: "(" "tag" "(*)" ")" ;
<tag-expr>:: <simple-tag> | <tag-set> | <tag-string> ;
<simple-tag>:: "(" <byte-string> <tag-expr>* ")" ;
<tag-set>:: "(" "*" "set" <tag-expr>* ")" ;
<tag-string>:: <byte-string> | <tag-range> | <tag-prefix> ;
<tag-prefix>:: "(" "*" "prefix" <byte-string> ")" ;
<tag-range>:: "(" "*" "range" <range-ordering> <low-lim>?
              <up-lim>? ")" ;
```

Reading this BNF for tags a number of questions arise:

- why does the <tag> definition differentiate between <tag-star> and "(""tag" <tag-expr>")" if <tag-star> is also "(""tag" ... ")" ?
- the name <tag-string> is somewhat illogical, it can be whether a byte string, a range or a prefix
- and the <tag-expr> is not very logical either; it can be a <tag-set> but also a <tag-range> or a <tag-prefix>, and all of them are "star-tags"

So we came up with another BNF that, we believe, is equivalent to the one defined in [SPKI] and should be clearer. The BNF definition is as follows:

```
<tag>:: "(" "tag" <tag-content> ")" ;
<tag-content>:: <tag-star> | <tag-expr> ;
<tag-star>:: "(*)" ;
<tag-expr>:: <tag-string> | <simple-tag> | <star-tag> ;
<tag-string>:: <byte-string> ;
<simple-tag>:: "(" <byte-string> <tag-expr>* ")" ;
<star-tag>:: <tag-set> | <tag-range> | <tag-prefix> ;
<tag-set>:: "(" "*" "set" <tag-expr>* ")" ;
<tag-range>:: "(" "*" "range" <range-ordering> <low-lim>?
              <up-lim>? ")" ;
<tag-prefix>:: "(" "*" "prefix" <byte-string> ")" ;
```

Even if the BNF has two extra definitions for dealing with tags, we believe this definition is structured in an easier way to understand it and that both definitions are equivalent. We introduce this

definition here because the XML Schema for tags is based upon it (see hereafter).

```
<xsd:element name="tag" type="tag-content"/>

<xsd:complexType name="tag-content">
 <xsd:choice>
  <xsd:group ref="tag-expression"/>
  <xsd:element ref="tag-null"/>
  <xsd:element ref="tag-star"/>
 </xsd:choice>
</xsd:complexType>

<xsd:element name="tag-star" type="tag-star"/>

<xsd:complexType name="tag-star"/>

<xsd:group name="tag-expression">
 <xsd:choice>
  <xsd:element ref="tag-string"/>
  <xsd:group ref="star-tag"/>
  <xsd:element ref="simple-tag"/>
 </xsd:choice>
</xsd:group>

<xsd:element name="tag-string" type="byte-string"/>

<xsd:element name="simple-tag" type="simple-tag"/>

<xsd:complexType name="simple-tag">
 <xsd:sequence>
  <xsd:element ref="type"/>
  <xsd:group ref="tag-expression"
             minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:group name="star-tag">
 <xsd:choice>
  <xsd:element ref="set"/>
  <xsd:element ref="prefix"/>
  <xsd:element ref="range"/>
 </xsd:choice>
</xsd:group>

<xsd:element name="set" type="tag-set"/>
```

```
<xsd:complexType name="tag-set">
 <xsd:sequence>
  <xsd:group ref="tag-expression"
             minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="range" type="tag-range"/>

<xsd:complexType name="tag-range">
 <xsd:sequence>
  <xsd:element name="range-ordering" type="range-ordering"/>
  <xsd:element name="low-lim" type="low-lim" minOccurs="0"/>
  <xsd:element name="up-lim" type="up-lim" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="prefix" type="byte-string"/>

<xsd:element name="tag-null" type="tag-null"/>

<xsd:complexType name="tag-null"/>
```

Note that we have included here the type <tag-null>. Although [SPKI] only defines this type to tuple-5 reduction, in some cases it may be useful to define it here. For example if the trust engine deals with XML document or one wants to "see" the reduction resulting tuple-5, even if it contains a null as tag.

4.9 <validity>

The validity specifies when a certificate is valid and/or online test information for the certificate.

Note that we have changed the tag defined in [SPKI] (<valid>) because it was prone to error. Precisely, the revalidation list object for certificates tags the list of valid certificates as <valid> too. It represents a small change to make things clearer.

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

<xsd:complexType name="validity">
 <xsd:sequence>
  <xsd:group ref="valid-basic"/>
  <xsd:element ref="online" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

```
<xsd:group name="valid-basic">
 <xsd:sequence>
  <xsd:element ref="not-before" minOccurs="0"/>
  <xsd:element ref="not-after" minOccurs="0"/>
 </xsd:sequence>
</xsd:group>

<xsd:element name="not-before" type="date"/>
<xsd:element name="not-after" type="date"/>
```

We define the type <validity> as a sequence of a <valid-basic> type, whose contents might be empty, and optionally, a list of online tests. In turn, the <valid-basic> type is a sequence of optional <not-before> and <not-after> types, which are self-explanatory.

4.9.1 <date>

```
<xsd:simpleType name="date">
 <xsd:restriction base="xsd:string">
  <xsd:whiteSpace value="collapse"/>
   <xsd:pattern value=
       '"[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[1-2][0-9]|3[0-1])_
       ([0-1][0-9]|2[0-4]):([0-5][0-9]):([0-5][0-9])"'/>
 </xsd:restriction>
</xsd:simpleType>
```

The definition of <date> states that a date is an extension of string, but with the restrictions imposed by the regular expression. This regular expression follows the date pattern as defined in [SPKI].

4.9.2 <online>

```
<xsd:element name="online" type="online-test"/>

<xsd:complexType name="online-test">
 <xsd:sequence>
  <xsd:element ref="online-type"/>
  <xsd:element ref="uris"/>
  <xsd:group ref="principal"/>
  <xsd:element ref="id"/>
  <xsd:group ref="spart" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

```
<xsd:element name="online-type" type="online-type"/>

<xsd:simpleType name="online-type" id="online-type">
 <xsd:restriction base="xsd:string">
  <xsd:whiteSpace value="collapse"/>
  <xsd:enumeration value="3:crl"/>
  <xsd:enumeration value="crl"/>
  <xsd:enumeration value="5:reval"/>
  <xsd:enumeration value="reval"/>
  <xsd:enumeration value="8:one-time"/>
  <xsd:enumeration value="one-time"/>
 </xsd:restriction>
</xsd:simpleType>
```

Until here, the definition of the online test should be rather clear. Things get more complex when defining the <new-cert> type of online tests. This online type breaks somewhat the general definition as presented above because it does not require principal neither parameters (the S-parts). In the definition hereafter, the type new-cert is defined as an extension to the base type <online-test> with the following restrictions: the online-type must be equal to <new-cert> and the fields <principal> and <spart> are not allowed. The definition of the type <online-test-new-cert> is as follows:

```
<xsd:complexType name="online-test-new-cert">
 <xsd:complexContent>
  <xsd:restriction base="online-test">
   <xsd:sequence>
    <xsd:element name="online-type">
     <xsd:simpleType>
         <xsd:restriction base="xsd:string">
       <xsd:whiteSpace value="collapse"/>
          <xsd:enumeration value="8:new-cert"/>
       <xsd:enumeration value="new-cert"/>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element ref="uris"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>
```

4.10 <comment>

```
<xsd:element name="comment" type="byte-string"/>
```

A comment is an optional field that is ignored by any processing code, but, probably, intended for reading by a human. A comment element is of type byte-string.

5 Name Certificate

Names are defined for human convenience. For actual trust engine computations, names must be reduced to principals. This section gives the XML Schema definition of a name certificate and a name.

Note that SPKI does not include an <issuer-info> option for a name certificate. The issuer needs no authorization in order to define a local name.

For the same reason, there is no "certification practice statement" for these name certificates. A name certificate implies nothing about the principals being named.

5.1 Name Certificate Definition

See section 4 for the abstract definition of a certificate. A name certificate is defined as follows:

```
<xsd:complexType name="name-cert">
 <xsd:complexContent>
  <xsd:restriction base="certificate">
   <xsd:sequence>
    <xsd:element ref="version" minOccurs="0"/>
    <xsd:element ref="display" minOccurs="0"/>
    <xsd:element ref="issuer-name"/>
    <xsd:element ref="subject"/>
    <xsd:element ref="validity" minOccurs="0"/>
    <xsd:element ref="comment" minOccurs="0"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="issuer-name" type="issuer-name"/>

<xsd:complexType name="issuer-name">
 <xsd:sequence>
  <xsd:group ref="principal"/>
  <xsd:element name="name" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>
```

Remember that, according to [SPKI], in a name certificate, the <tag> field is omitted and <tag-star> is assumed. Also there is no <propagate> field. A name definition is like a cord, passing everything the name is granted through to the subject.

5.2 <name>

A <name> element is an option for <subject>, when one wants to generate a certificate granting authorization to either a named group of principals or to a principal that has not yet been defined, or whose binding with a name might change.

This name can be either relative with the principal implied being the issuer of the certificate, or fully-qualified in which the principal is explicitly specified. The later allows the issuer of the certificate to make references to any other namespace.

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

<xsd:complexType name="name">
 <xsd:choice>
  <xsd:group ref="fq-name" />
  <xsd:group ref="relative-name"/>
 </xsd:choice>
</xsd:complexType>

<xsd:group name="fq-name">
 <xsd:sequence>
  <xsd:group ref="principal" />
  <xsd:element ref="local-name" maxOccurs="unbounded" />
 </xsd:sequence>
</xsd:group>

<xsd:group name="relative-name">
 <xsd:sequence>
  <xsd:element ref="local-name" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:group>

<xsd:element name="local-name" type="byte-string"/>
```

6 ACLs and Sequences

ACL and sequence structures are in the gray area of SPKI. ACLs are private to the verifier and thus specific to one developer or application. Sequences, on the other hand, can be thought of as part of the protocol using certificates.

6.1 <acl>

An ACL is a list of assertions: certificates bodies that do not need issuer fields or signatures because they should be being held locally to the verifier in secure memory. The fields of an ACL are the same fields of an authorization certificate; therefore we will not repeat their definition here. Since an ACL is not communicated to anyone, developers are free to choose their own formats.

An ACL is defined as follows:

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

<xsd:complexType name="acl">
 <xsd:sequence>
  <xsd:element ref="version" minOccurs="0"/>
  <xsd:element ref="entry" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="entry" type="acl-entry"/>

<xsd:complexType name="acl-entry">
 <xsd:sequence>
  <xsd:group ref="subject-obj"/>
  <xsd:element ref="propagate" minOccurs="0"/>
  <xsd:element ref="tag"/>
  <xsd:element ref="validity" minOccurs="0"/>
  <xsd:element ref="comment" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

6.2 <sequence>

A sequence is an ordered list of objects that the verifier is to consider when deciding to grant access. By reducing the certificates in the sequence that the final subject has been granted authority through the sequence. For details on sequence reduction see section 8 in [SPKI] and section 6 in [RFC2693].

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

   <xsd:complexType name="sequence">
    <xsd:sequence maxOccurs="unbounded">
     <xsd:group ref="seq-entry"/>
    <xsd:sequence/>
   <xsd:complexType/>

   <xsd:group name="seq-entry">
    <xsd:choice>
     <xsd:element ref="cert"/>
     <xsd:element ref="public-key"/>
     <xsd:element ref="signature"/>
     <xsd:element ref="do-hash"/>
     <xsd:element ref="do"/>
     <xsd:element ref="reval"/>
     <xsd:element ref="crl"/>
     <xsd:element ref="delta-crl"/>
    </xsd:choice>
   </xsd:group>
```

7 Online Test Reply Formats

An online test results in a digitally signed object carrying its own date range, explicitly or implicitly. The object specifies either a list of invalid certificates or that a given certificate (or list of certificates) is still valid.

Each of these objects contains a validity period interval. The object is valid only during that interval and a sequence of objects must be issued for non-overlapping intervals, so that the user of the object knows when it has the current one.

7.1 CRL and detla-CRL

A CRL is a list of certificates that have been revoked (are no longer valid). If one wants to provide CRLs, and that CRL grows, one may prefer to send only a delta-CRL.

```
<xsd:complexType name="crl">
 <xsd:sequence>
  <xsd:element ref="version" minOccurs="0"/>
  <xsd:element ref="canceled"/>
  <xsd:group ref="valid-basic"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="delta-crl">
 <xsd:sequence>
  <xsd:element ref="version" minOccurs="0"/>
  <xsd:element ref="hash"/>
  <xsd:element ref="canceled"/>
  <xsd:group ref="valid-basic"/>
 </xsd:sequence>
</xsd:complexType>
```

The hash in the delta-crl is the hash of the predecessor CRL or delta-CRL that this one is modifying. For convenience, the hash should probably also have a URI pointing the user to that predecessor.

7.2 Revalidation and One-time Revalidation

A revalidation is the logical opposite of a CRL. Instead of listing the certificates that are no longer valid, the revalidation instrument gives the list of certificates that are valid.

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

   <xsd:complexType name="reval">
    <xsd:sequence>
     <xsd:element ref="version" minOccurs="0"/>
     <xsd:element ref="valid"/>
     <xsd:group ref="reval-body"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="valid" type="reval-hash-list"/>

   <xsd:group name="reval-body">
    <xsd:choice>
     <xsd:element ref="one-time"/>
     <xsd:group ref="valid-basic"/>
    </xsd:choice>
   </xsd:group>

   <xsd:complexType name="reval-hash-list">
    <xsd:sequence>
     <xsd:element ref="hash" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
```

8 From S-expressions to XML-encoded Certificates

There are a number of alternatives for dealing with the trans-coding from S-expressions to XML-based SPKI objects. This process corresponds to what in XML terms is called upstream process: process for expressing information abstractly. This upstream process can be seen as the generation, from raw information, a XML document. Its counterpart is the downstream process, the process by which information is processed and/or consumed.

XML and associated standards do not address the upstream process, since this process needs to deal with "unstructured and raw" information from an XML point of view. Thus there is no XML standard way to address this issue and each developer or application should come up with its own solution.

8.1 Adapted S-expressions Parser

A first approach for this process in the case of SPKI S-expressions is the one taken in [S2X] by Carl Ellison. The piece of software parses an S-expression, and when generating the XML output, it recognizes the SPKI keywords and treats them in order to generate the equivalent XML document. If one pushed this approach further, we could have an SPKI library written in any language that parses SPKI objects encoded in S-expressions and translates the in-memory object-tree to XML according to the syntax defined by the XML Schema.

Taking into account that the XML Schema defined in this document specifies a syntax and grammar that matches (almost) that of [SPKI], it should not be too difficult to adapt Carl Ellison's code in [S2X] in order to provide an output conforming to the SPKI-XML Schema.

8.2 Custom Compiler

Given that in [SPKI] we have a pseudo-formal BNF syntax of all SPKI objects, one could use it in order to generate a compiler that reads and parses SPKI S-expressions and generates the corresponding XML-SPKI document. These are well-known techniques in the world of language theory and compilers. A developer could use tools such as LEX and YACC, or JavaCC among many others.

8.3 Other Possibilities

The approaches presented above are not the only ones. For example, Sun is defining the "Java Architecture for XML Binding" [JAXB] through its "Community Process". Its goal is to write a code generator from a XML Schema or DTD so that the components of an XML document are mapped to in-memory objects that represent, in an

obvious and useful way, the document's intended meaning according to its schema.

A developer could take the set of classes generated by JAXB compilers and write the code for unmarshalling from S-expressions instead of XML documents. Once this has been achieved, calling the marshalling method on the root element of the object hierarchy would generate the corresponding validated XML document.

It is not our intention here to provide an exhaustive list of all different approaches for dealing with the generation of an XML document from a S-expression. We have simply pointed some of the alternatives. Each developer should choose what approach to follow, maybe one of the mentioned in this section, maybe another one.

9 From XML to S-expressions Encoded Certificates

In previous sections we have already presented and discussed the XML representation of SPKI certificates, and gave some hints for the trans-coding from S-expressions to XML-based encoded SPKI objects. In order to achieve total transparency and independence of the encoding form, we need to address the trans-coding from XML to S-expressions encoded SPKI objects. The objective of this section is to define the rules for the transformation from XML to S-expression encoded SPKI objects. How these rules are applied is another issue, for which we describe some options.

9.1 Why XSL and XSLT?

There are two basic approaches to dealing with trans-coding from XML to S-expression encoded SPKI objects: choose a procedurally-oriented language and implement the rules for the trans-coding, or define those rules through XSLT and leave the application of the rules open.

Choosing a procedurally-oriented programming language solves the issue, but has some implications that might not me desirable:

- the user is forced to choose, if available, one implementation among several
- the definition of the transformation rules and the application are fixed by the concrete implementation
- the user cannot choose how the rules should be applied
- developers need to deal with all low-level programming aspects of a procedurally-oriented language, such as memory management or node manipulation, with all the implementation errors that might appear.

On the other hand, XSLT focuses on the definition of rules for the transformation of XML documents, mainly for rendering but also for format transformation. Some of its advantages are:

- allows to deal with the high-level aspects of transformations such as sorting, counting or matching, leaving the low-level ones to the XSLT processor
- the declarative nature of stylesheet markup makes XSLT so very much more accessible to non-programmers than the imperative nature of procedurally-oriented transformation languages
- there is a strong relation between XML and XSLT that we can take advantage of in order to ease these transformations, which can be seen as an XML downstream process
- as consequence, we can bind an XML document with its transformation via process instructions inside the XML document. These process instructions instruct the browser on how to display the document

- the extensible nature of XSLT lets transformations that cannot be expressed by the XSLT "conforming techniques" [XSLT] can be so through processor extensions.

9.2 Application of Rules

As said earlier, in this section we focus on the definition of rules for the trans-coding of SPKI object, from XML to S-expressions. There are a number of ways these rules can be applied, which we introduce here. We believe defining the transformation rules and leave their application undefined is the most flexible and open solution to this issue.

A user with the SPKI-XSLT can transform SPKI-XML document via a XSLT processor. An XSLT processor takes as input a XML document and the XSLT stylesheet and produces a text output that represents the transformed document. There are a number of XSLT processors available today, being the best known Xalan and XT.

If the transformation is oriented to the rendering of the XML document, one can associate the stylesheet defining how to display the XML document to the document. The displaying application, for example a browser, will interpret the transformation rules and render it transparently.

Another alternative oriented to those cases in which a piece of software doing the transformation process is required, would imply the use of the rules definition and compile it into lightweight and portable codes, called Translets. Translets can be used from, for example, a Java component to process XML documents against these compiled stylesheets and generate output according to the stylesheet instructions. Translets have some advantages over the above alternatives: they have small footprint, they are fast and provide high throughput, they do not require runtime compilation and provide protection of intellectual property if the rules require it.

9.3 Limitations in XSLT 1.0

XSLT was originally designed primarily for transforming XML vocabularies to the XSL formatting vocabulary. This does not preclude us from using XSLT for other transformation requirements. For this reason, the designers do not claim XSLT 1.0 is a general-purpose transformation language.

Given that we use XSLT for a downstream process to a non-XSL formatting vocabulary, S-expressions in canonical form, it is understandable we find some limitations. XSLT would need to support the decoding of hexadecimal, octal and base-64 for transforming a non-printable byte-string onto its canonical form. For these reasons, the output of the XSLY stylesheet provided in this

document generates S-expressions in advanced form.

This limitation could be solved using XSLT processor extensions. But this version of XSLT does not provide standard mechanisms for defining implementations of extensions. Therefore, an XSLT Stylesheet that must be portable across XSLT processors cannot rely on particular extensions being available.

XSLT 1.1 introduces an additional data-type into the expression language. This additional data is called external object. An external object is created by an external programming language and returned by an extension function. This external object will be able to resolve all of the coding/decoding problems.

Furthermore XSLT 1.1 also supports extension functions definitions. The top-level xsl:script elements provide an implementation of extensions functions in a particular namespace. Coding/decoding to/from base-64/hexadecimal/octal can be solved via a definition in some language (currently some ECMAScript scripting languages). Its bindings also allow the function implementations to be provided directly in the content.

As of this writing, there is not any XSLT processor supporting XSLT version 1.1

9.4 XSLT Rules Definition

We can characterize XSLT from other techniques for transforming information by regarding it as a "transformation by example", differentiating many other techniques as "transformation by program logic". This perspective focuses on the distinction that our application is not to tell a XSLT processor how to effect the changes we need, but rather we tell the XSLT processor what we want as result, and it is the processor's responsibility to do the work.

The XSLT recommendation provides a vocabulary for specifying templates that function as "examples of result". Based on who we instruct the XSLT processor to access the source of the data being transformed, the processor will incrementally build the result by adding the filled-in templates.

A XSLT stylesheet defines a set of rules (via xsl:template) that are to be applied on a XML document. A XSLT processor parses the XML document looking for templates defined in the rules, "consumes" the matched element and generates a result-tree. A priory, XSLT only identifies four types of objects from an XML document:

- the root element of the XML document
- nodes in the tree representing the XML document
- text that are the leaves of the tree representing the XML document
- element attributes that are part too of the tree

- comments, for which one can specify a specific treatment
- process instructions, for which, again, one can specify specific treatment

A template may contain both text that will appear literally in the output document and XSL instructions that copy data from the input XML document to the result. Because all XSL instructions are in the xsl: namespace, one can easily distinguish between the elements that are literal data to be copied to the output and XSL instructions.

9.4.1 XSLT for SPKI-XML to S-expressions

The principle in the rules definition is very simple: we define a rule for each element that is to generate an output. These elements correspond to types S-expressions; for example '(public-key'. Then we define a specific treatment for terminal text nodes. All byte-strings will go through this template and it is here we verify whether it has a display-type or not. This approach works because we only need to deal with one attribute (the display-type) for generating S-expressions, and we ignore comments and process instructions.

We see a bit more in detail the rules definition. For the complete XSLT Stylesheet see Appendix D.

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

The above rule "turns on" the processor. It matches the root element and starts the process by instructing the XSLT processor to apply the templates on the other elements of the tree.

```
<xsl:template match="public-key">
    <xsl:text>(public-key </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>)</xsl:text>
</xsl:template>
```

This is the rule for public keys: when the XSLT processor finds an element 'public-key' it produces the output '(public-key ', re-applies the templates on the next element in the tree, and produces some more output ')'. This process is similar to the generation of S-expressions: first the opening '(' and the type, then all of its parts, and finally the closing ')'.

```
<xsl:template match="text()">
    <xsl:variable name="dt">
        <xsl:value-of select="parent::*/@display-type"/>
    </xsl:variable>
    <xsl:text>  </xsl:text>
    <xsl:if test="string-length($dt) != 0">
        <xsl:value-of select="$dt"/>
    </xsl:if>
    <xsl:value-of select="."/>
</xsl:template>
```

This template is used by the processor when it finds a text note (match="text()"). When the processor finds a text, it will go back to the parent node and assign its display-type attribute to the variable 'dt'. This is type of tree access is defined by XPath, that provides direct access to document elements. Then we verify whether 'dt' has some value, the display-type, and if so it is copied to output. In any case we copy the text value.

With these simple rules we are able to transform XML-encoded SPKI certificates back to S-expressions. Here, these S-expressions are in advanced form, but in the future we expect it will be possible to directly generate the canonical form of S-expressions.

10 Open Issues in XML-encoded SPKI Certificates

There are a number of issues, problems or alternatives in the XML-SPKI Schema presented and discussed in this document. In this section we try to present all of them in order to open discussions and reach an agreement with the SPKI community.

The order of the different points presented here does not follow any specific criteria, except for its pseudo-randomness.

10.1 XML-DSIG

XML-DSIG [XSIG] is a joint effort between W3C and IETF in order to specify the syntax and processing rules for creating and representing digital signatures in XML. XML-DSIG defines the KeyInfo element in a signature, which is an optional element that enables the verifier to obtain the key needed to verify the signature. If omitted, it is assumed that the verifier is able to identify the key based on the application context. The KeyInfo element may have children elements, for example PGPData, SPKIData or X509Data. Semantically, these should be seen as data structures that somehow "empower" or authenticate the signing key.

J. Paajarvi in [PAAJ] uses XML signatures to represent signatures over SPKI objects. However we do not believe this is the best option in this case for the following reasons:

- The contents of KeyInfo should provide the necessary information for authenticating the signing key. In the case of SPKI this should be a sequence, in SPKI terms, going from the verifiers ACL to the signing key. However, SPKI signatures cannot contain such information since they are only meaningful regarding the previous element in the sequence.

- Trans-coding from S-expression based signatures to XML and vice-versa would represent a rather complex definition because we would need also to provide key values as expected by XML-DSIG, and these are not always compatible with the objects defined by SPKI. Furthermore it would break the syntax and semantics of a SPKI signature.

- Interoperability between S-expression and XML signatures (see sub-section hereafter) becomes impossible to achieve, at least in the case of "nested" signatures (signing signed objects).

For these reasons, we decided to define XML SPKI signatures in such a way that they fully follow their definition in [SPKI] both syntactically and semantically.

10.2 Signature Level Interoperability and XML Canonical Forms

The goal of this document is not only the definition of an XML Schema for SPKI certificates, but rather provide a way for the seamless usage of SPKI encoded in different formats, namely S-expressions and XML. Ideally, one should be able to generate a sequence fully from XML, with signature elements in it, tans-code it to S-expressions and be able to verify the signature. This is the only way to achieve full interoperability between S-expression and XML-encoded SPKI signatures.

The only solution in order to achieve this would be to define a XML canonicalization method such that the resulting canonical form of the XML document matches that of the canonical S-expression. There may be the need for some processing before applying the canonicalization, for example character encoding rules.

Today this seems rather unlikely: to start with, we are not yet able to define, with XML-based standards, the trans-coding from XML-SPKI to S-expressions in their canonical form. But, as exposed in previous section, when XSLT processors implementing version 1.1 of XSLT are available, this step will be possible.

If, and this is a big if, one could specify as canonicalization method, the application of a XSLT stylesheet identified by an URI, and take the result as canonical form for the document, we would achieve total interoperability between XML and S-expressions encoded SPKI objects.

Note that just because this has not been done until today does not mean that it will never be done. XSL and XSLT are rather new in terms of XML history. If this approach is found interesting for the objectives of SPKI, maybe SPKI as an IETF Working Group or individuals could propose it to both XML Canonicalization and XML Signature groups.

10.3 Any Namespace, Why Not Allowed?

As we have already pointed out previously in this document, there are a number of places (key values and signatures) in which we could use references to other namespaces (##any in XML) in order to define new types of data (new key values and signatures). Doing so one could extend the SPKI specification provided with this schema simply by defining the new objects in another, probably custom, namespace. We could still validate the XML document and ensure that the externally-defined data structures are well-formed (but not they are valid). But the case of SPKI we cannot allow it for one reason: we want to be able to transform any data structure in the document back to S-expressions. And in order to do so, the stylesheet needs to "know" the XML document in order to transform them to S-expressions.

The same restriction applies to private key values, hashes and signatures.

10.4 String or <byte-string> for Identifiers

In the specification process we choose to use strings as algorithm identifiers in order to simplify the algorithm identifier types. In case this shortcut is invalid, we would need to define algorithm identifiers as a union complex type between <bytes> (for known algorithms) and <string> (for non-standard algorithms).

If even <bytes> is still unacceptable, then the only solution is to define algorithms identifiers as <byte-string>, but in that case, we would be unable to enumerate them.

This is an issue open to discussion.

10.5 Limitations in XML Sets xsd:all

Sets in XML (xsd:all) are the only group model in that can implement unordered elements. However it imposes some important restrictions as of their contents:

- all sets must be content-local and the top-level element of their type.
- all sets may contain only individual element declarations

Furthermore, most parsers we have tested do not correctly process the xsd:all group model.

For these reasons we had to impose order in both certificates and ACLs. We expect this problem to be solves in future releases of XML Schema and parser implementations.

10.6 Transforms Element for <byte-string>

As we already pointed out in section 3.1.1, the <bytes> type does not support the transformations attribute. These attributes specify what are the transformations to apply on a piece of data in a XML document in order to obtain the original value. We could use it to indicate the encoding of a <byte-string>, for example base-64. In case the SPKI community finds this useful, this slight modification can be added in next version of the XML-SPKI Schema. It is open to discussion.

10.7 XML Derived Types

XML allows the definition of a type as extension or restriction to another already-defined type. We have used this mechanism for certificates (authorization and name) and for online new-cert construct.

In order to make this feature of derivation in XML Schemas work, and to identify exactly which derived type is intended in a specific case, the derived type must be identified in the instance document. The derived type is identified using the xsi:type attribute, which is part of the XML Schema instance namespace.

10.8 XML Schema 2001-05-02

The W3C released a new version of XML Schema Technical Recommendation as of March 2nd 2001. The XML-SPKI schema presented in this document does not follow that recommendation. There are a number of significant changes in several aspects that we have not yet applied. These changes affect the patterns, character encoding and pre-defined types (xsd:binary no longer exists and has been substituted by several xsd:XXXbinary).

In next version of this document we will apply these changes.

Appendix A - Examples of XML-encoded SPKI objects

Here we provide the XML form of some of the examples that appear in [CERTEX], plus some others. For each example we provide the advanced S-expression form and the corresponding XML document.

A.1 FTP Example

This example corresponds to section 2.1 in [CERTEX]. The tag gives permission to FTP host cybercash.com and login as user cme.

```
(tag (ftp cybercash.com cme))
```

And the corresponding XML element is:

```
<tag>
 <simple-tag>
  <type>ftp</type>
  <tag-string>cybercash.com</tag-string>
  <tag-string>cme</tag-string>
 </simple-tag>
</tag>
```

A.2 HTTP Example

This example is based on section 2.2 in [CERTEX]. The tag gives permission to access the web pages at the specified URI.

```
(tag (http (* prefix http://acme.com/company-private/personnel/)))
```

And the corresponding XML element is:

```
<tag>
 <simple-tag>
  <type>http</type>
  <prefix>http://acme.com/company-private/personnel/</prefix>
 </simple-tag>
</tag>
```

A.3 Spend Money Example

This example corresponds to section 2.5 in [CERTEX]. The tag gives permission to spend up to 500.00 per electronic check from the indicated checking account at the Bank of Boston. We have modified the example in order to conform with the tag specification.

```
(tag (spend BankBoston "011000390 436 20608"
  (* range numeric le "500.00")))
```

And the corresponding XML element is:

```
<tag>
 <simple-tag>
  <type>spend</type>
  <tag-string>BankBoston</tag-string>
  <tag-string>"011000390 436 20608"</tag-string>
  <range>
   <range-ordering>numeric</range-ordering>
   <up-lim>
    <limit-type>le</limit-type>
    <limit-value>"500.00"</limit-value>
   </up-lim>
  </range>
 </simple-tag>
</tag>
```

A.4 Locator Certificate Example

This example corresponds to section 3.1 in [CERTEX]. It represents a locator certificate, that is, a certificate issued by a company that promises to keep track of the indicated keyholder until the not-after date, and promises to serve the keyholder with papers for the indicated fee in the indicated currency, up until the not-after date.

```
(cert
  (issuer (hash md5 |u2kl73MiObh5o1zkGmHdbA==|))
  (subject (keyholder (hash md5 |kuXyqx8jYWdZ/j7Vffr+yg==| )))
  (tag (tracking-fee "150" USD))
  (validity (not-after "2003-01-01_00:00:00")))
```

And the corresponding XML-encoded certificate is:

```
<cert xsi:type="authorization-cert">
 <issuer>
  <hash>
   <hash-alg-name>md5</hash-alg-name>
   <hash-value>|u2kl73MiObh5o1zkGmHdbA==|</hash-value>
  </hash>
 </issuer>
 <subject>
  <keyholder>
   <hash>
    <hash-alg-name>md5</hash-alg-name>
    <hash-value>|kuXyqx8jYWdZ/j7Vffr+yg==|</hash-value>
   </hash>
  </keyholder>
 </subject>
 <propagate/>
```

```
    <tag>
     <simple-tag>
      <type>tracking-fee</type>
      <tag-string>"150"</tag-string>
      <tag-string>USD</tag-string>
     </simple-tag>
    </tag>
    <validity>
     <not-after>"2003-01-01_00:00:00"</not-after>
    </validity>
   </cert>
```

A.5 Insurance Certificate Example

This example corresponds to section 3.2 in [CERTEX]. Instead of tracking down a keyholder and serving papers on him or her, the person relying on a certificate might prefer that some insurance company pay the penalty amount in the event of non-performance, and then worry on its own about collecting that fee (plus damages, no doubt) from the keyholder.

```
(cert
  (issuer (hash md5 |u2kl73MiObh5o1zkGmHdbA==|))
  (subject (keyholder (hash md5 |kuXyqx8jYWdZ/j7Vffr+yg==| )))
  (tag
    (insured
    (amount "50000" USD)
    (to (hash md5 |1r8ICXryJw6v/B4MQdTU/Q==|))
    (for "Failure to perform under contract (on file): "
      (hash md5 |gPA50iM6yETsixLgo2kVlA==|))))
  (validity (not-after "2003-01-01_00:00:00")))
```

And the corresponding XML-encoded certificate is:

```
<cert xsi:type="authorization-cert">
 <issuer>
  <hash>
   <hash-alg-name>3:md5</hash-alg-name>
   <hash-value>|u2kl73MiObh5o1zkGmHdbA==|</hash-value>
  </hash>
 </issuer>
 <subject>
  <keyholder>
   <hash>
    <hash-alg-name>3:md5</hash-alg-name>
    <hash-value>|kuXyqx8jYWdZ/j7Vffr+yg==|</hash-value>
   </hash>
  </keyholder>
 </subject>
 <propagate />
```

```
    <tag>
     <simple-tag>
      <type>7:insured</type>
      <simple-tag>
       <type>6:amount</type>
       <tag-string>5:50000</tag-string>
       <tag-string>3:USD</tag-string>
      </simple-tag>
      <simple-tag>
       <type>2:to</type>
       <simple-tag>
        <type>4:hash</type>
        <tag-string>3:md5</tag-string>
        <tag-string>|1r8ICXryJw6v/B4MQdTU/Q==|</tag-string>
       </simple-tag>
      </simple-tag>
      <simple-tag>
       <type>3:for</type>
       <tag-string>45:Failure to perform under contract
                  (on file):</tag-string>
       <simple-tag>
        <type>4:hash</type>
        <tag-string>3:md5</tag-string>
        <tag-string>|gPA50iM6yETsixLgo2kVlA==|</tag-string>
       </simple-tag>
      </simple-tag>
     </simple-tag>
    </tag>
    <validity>
     <not-after>"2003-01-01_00:00:00"</not-after>
    </validity>
   </cert>
```

A.6 Auto Certificate Example

This example corresponds to section 3.3 in [CERTEX].

```
(cert
  (issuer (hash sha1 |1QvsTPF0/vqHPGODX/yEN8ro+sc=|))
  (subject (keyholder (hash sha1 |1QvsTPF0/vqHPGODX/yEN8ro+sc=|)))
  (tag
    (* set
      (name "Carl")
      (e-mail "cme@acm.org"))))
```

And the corresponding XML-encoded certificate is:

```
<cert xsi:type="authorization-cert">
 <issuer>
  <hash>
   <hash-alg-name>sha1</hash-alg-name>
   <hash-value>|1QvsTPF0/vqHPGODX/yEN8ro+sc=|</hash-value>
  </hash>
 </issuer>
 <subject>
  <keyholder>
   <hash>
    <hash-alg-name>sha1</hash-alg-name>
    <hash-value>|1QvsTPF0/vqHPGODX/yEN8ro+sc=|</hash-value>
   </hash>
  </keyholder>
 </subject>
 <tag>
  <set>
   <simple-tag>
    <type>name</type>
    <tag-string>Carl</tag-string>
   </simple-tag>
   <simple-tag>
    <type>e-mail</type>
    <tag-string>"cme@acm.org"</tag-string>
   </simple-tag>
  </set>
 </tag>
 <validity>
  <not-after>"2003-01-01_00:00:00"</not-after>
 </validity>
</cert>
```

Appendix B - Full SPKI-XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
              version="1.0">
<xsd:include schemaLocation="sexpr.xsd"/>

<!--                                                              -->
<!--             XML Schema for SPKI certificates                 -->
<!--             --------------------------------                 -->
<!-- Based on "Simple Public Key Certificates"                    -->
<!--                     <draft-ietf-spki-cert-structure-06.txt>  -->
<!-- Description in "XML-SPKI Certificate Strucutre"              -->
<!--                     <draft-orri-xml-spki-cert-struc-00.txt>  -->
<!-- Note: references to section numbers correspond to section    -->
<!-- numbers in document "XML-SPKI Certificate Structure"         -->

<!--              TYPE AND ELEMENT DEFINITIONS                    -->
<!--             ==============================                   -->
<!--                                                              -->
<!-- Section 3.2: Primitive Objects                               -->
<!--                                                              -->
<!-- Section 3.2.1: Public Keys                                   -->
<!--                                                              -->

<xsd:group name="principal">
 <xsd:choice>
  <xsd:element ref="public-key"/>
  <xsd:element ref="hash"/>
 </xsd:choice>
</xsd:group>

<xsd:complexType name="principal">
 <xsd:group ref="principal"/>
</xsd:complexType>

<xsd:element name="public-key" type="pub-key"/>

<xsd:complexType name="pub-key">
 <xsd:sequence>
  <xsd:element ref="key-value"/>
  <xsd:element ref="uris" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

```
   <xsd:element name="key-value" type="key-value"/>

   <xsd:complexType name="key-value">
    <xsd:sequence>
     <xsd:element ref="pub-sig-alg-id"/>
     <xsd:choice>
      <xsd:element ref="rsa-key-value"/>
      <xsd:element ref="dsa-key-value"/>
      <xsd:element ref="any-key-value"/>
     </xsd:choice>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="pub-sig-alg-id" type="pub-sig-alg-id"/>

   <xsd:simpleType name="pub-sig-alg-id">
    <xsd:union memberTypes="std-pub-sig-alg-id xsd:string"/>
   </xsd:simpleType>

   <xsd:simpleType name="std-pub-sig-alg-id">
    <xsd:restriction base="xsd:string">
     <xsd:whiteSpace value="collapse"/>
     <xsd:enumeration value="13:rsa-pkcs1-md5"/>
     <xsd:enumeration value="rsa-pkcs1-md5"/>
     <xsd:enumeration value="14:rsa-pkcs1-sha1"/>
     <xsd:enumeration value="rsa-pkcs1-sha1"/>
     <xsd:enumeration value="9:rsa-pkcs1"/>
     <xsd:enumeration value="rsa-pkcs1"/>
     <xsd:enumeration value="8:dsa-sha1"/>
     <xsd:enumeration value="dsa-sha1"/>
    </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="rsa-key-value" type="rsa-key-value"/>

   <xsd:complexType name="rsa-key-value">
    <xsd:sequence>
     <xsd:element name="e" type="byte-string"/>
     <xsd:element name="n" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="dsa-key-value" type="dsa-key-value"/>

   <xsd:complexType name="dsa-key-value">
    <xsd:sequence>
     <xsd:element name="p" type="byte-string"/>
     <xsd:element name="g" type="byte-string" minOccurs="0"/>
     <xsd:element name="q" type="byte-string"/>
     <xsd:element name="y" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>
```

```
   <xsd:element name="any-key-value" type="any-key-value"/>

   <xsd:complexType name="any-key-value">
    <xsd:sequence>
     <xsd:element ref="sexpr" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <!--                                                              -->
   <!-- Section 3.2.2: Private Keys                                  -->
   <!--                                                              -->

   <xsd:element name="private-key" type="priv-key"/>

   <xsd:complexType name="priv-key">
    <xsd:sequence>
     <xsd:element ref="pub-sig-alg-id"/>
     <xsd:choice>
      <xsd:element ref="rsa-priv-key-value"/>
      <xsd:element ref="rsa-crt-key-value"/>
      <xsd:element ref="dsa-priv-key-value"/>
      <xsd:element ref="any-key-value"/>
     </xsd:choice>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="rsa-priv-key-value" type="rsa-priv-key-value"/>

   <xsd:complexType name="rsa-priv-key-value">
    <xsd:sequence>
     <xsd:element name="e" type="byte-string" minOccurs="0"/>
     <xsd:element name="n" type="byte-string"/>
     <xsd:element name="d" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="rsa-crt-key-value" type="rsa-crt-key-value"/>

   <xsd:complexType name="rsa-crt-key-value">
    <xsd:sequence>
     <xsd:element name="e" type="byte-string" minOccurs="0"/>
     <xsd:element name="n" type="byte-string"/>
     <xsd:element name="d" type="byte-string"/>
     <xsd:element name="p" type="byte-string"/>
     <xsd:element name="q" type="byte-string"/>
     <xsd:element name="a" type="byte-string"/>
     <xsd:element name="b" type="byte-string"/>
     <xsd:element name="c" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>
```

```
   <xsd:element name="dsa-priv-key-value" type="dsa-priv-key-value"/>

   <xsd:complexType name="dsa-priv-key-value">
    <xsd:sequence>
     <xsd:element name="p" type="byte-string"/>
     <xsd:element name="g" type="byte-string" minOccurs="0"/>
     <xsd:element name="q" type="byte-string"/>
     <xsd:element name="y" type="byte-string"/>
     <xsd:element name="x" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>

   <!--                                                              -->
   <!-- Section 3.2.3: Hashes                                        -->
   <!--                                                              -->

   <xsd:element name="hash" type="hash"/>

   <xsd:complexType name="hash">
    <xsd:sequence>
     <xsd:element ref="hash-alg-name"/>
     <xsd:element ref="hash-value"/>
     <xsd:element ref="uris" minOccurs="0"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="hash-alg-name" type="hash-alg-name"/>

   <xsd:simpleType name="hash-alg-name">
    <xsd:union memberTypes="std-hash-alg-name xsd:string"/>
   </xsd:simpleType>

   <xsd:simpleType name="std-hash-alg-name">
    <xsd:restriction base="xsd:string">
     <xsd:enumeration value="3:md5"/>
     <xsd:enumeration value="md5"/>
     <xsd:enumeration value="4:sha1"/>
     <xsd:enumeration value="sha1"/>
    </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="hash-value" type="byte-string"/>
```

```
   <!--                                                              -->
   <!-- Section 3.2.4: Signatures                                    -->
   <!--                                                              -->

   <xsd:element name="signature" type="signature"/>

   <xsd:complexType name="signature">
    <xsd:sequence>
     <xsd:element ref="hash"/>
     <xsd:group ref="principal"/>
     <xsd:element ref="signature-value"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="signature-value" type="signature-value"/>

   <xsd:complexType name="signature-value">
    <xsd:sequence>
     <xsd:element ref="pub-sig-alg-id"/>
     <xsd:group ref="sig-params"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:group name="sig-params">
    <xsd:choice>
     <xsd:element ref="dsa-sig-params"/>
     <xsd:element ref="rsa-sig-params"/>
     <xsd:element ref="any-sig-params"/>
    </xsd:choice>
   </xsd:group>

   <xsd:element name="rsa-sig-params" type="byte-string"/>

   <xsd:element name="dsa-sig-params" type="dsa-sig-params"/>

   <xsd:complexType name="dsa-sig-params">
    <xsd:sequence>
     <xsd:element name="r" type="byte-string"/>
     <xsd:element name="s" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="any-sig-params" type="any-sig-params"/>

   <xsd:complexType name="any-sig-params">
    <xsd:choice>
     <xsd:element ref="byte-string"/>
     <xsd:element ref="sexpr" maxOccurs="unbounded"/>
    </xsd:choice>
   </xsd:complexType>
```

```
<!--                                                                -->
<!-- Section 4: Authorization Certificates                          -->
<!--                                                                -->

<xsd:element name="cert" type="certificate"/>

<xsd:complexType name="certificate">
 <xsd:sequence>
  <xsd:element ref="version" minOccurs="0"/>
  <xsd:element ref="display" minOccurs="0"/>
  <xsd:element ref="subject"/>
  <xsd:element ref="validity" minOccurs="0"/>
  <xsd:element ref="comment" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="authorization-cert">
 <xsd:complexContent>
  <xsd:restriction base="certificate">
   <xsd:sequence>
    <xsd:element ref="version" minOccurs="0"/>
    <xsd:element ref="display" minOccurs="0"/>
    <xsd:element ref="issuer"/>
    <xsd:element ref="issuer-info" minOccurs="0"/>
    <xsd:element ref="subject"/>
    <xsd:element ref="subject-info" minOccurs="0"/>
    <xsd:element ref="propagate" minOccurs="0"/>
    <xsd:element ref="tag"/>
    <xsd:element ref="validity" minOccurs="0"/>
    <xsd:element ref="comment" minOccurs="0"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

<!--                                                                -->
<!-- Section 4.X: Parts of Authorization Certificates               -->
<!--                                                                -->

<xsd:element name="version" type="integer"/>

<xsd:element name="display" type="byte-string"/>

<xsd:element name="issuer" type="principal"/>

<xsd:element name="issuer-info" type="uris"/>

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

```
   <xsd:complexType name="uris">
    <xsd:sequence>
     <xsd:element ref="uri" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="uri" type="byte-string"/>

   <xsd:element name="subject-info" type="uris"/>

   <xsd:element name="propagate" type="deleg"/>

   <xsd:complexType name="deleg"/>

   <xsd:element name="comment" type="byte-string"/>

   <!--                                                                -->
   <!-- Section 4.5: Subject and Subject Objects                       -->
   <!--                                                                -->

   <xsd:element name="subject" type="subj-obj"/>

   <xsd:complexType name="subj-obj">
    <xsd:group ref="subject-obj"/>
   </xsd:complexType>

   <xsd:group name="subject-obj">
    <xsd:choice>
     <xsd:group ref="principal"/>
     <xsd:element ref="name"/>
     <xsd:element ref="object-hash"/>
     <xsd:element ref="keyholder"/>
     <xsd:element ref="k-of-n"/>
    </xsd:choice>
   </xsd:group>

   <!-- <name> see section 5                                          -->

   <xsd:element name="object-hash" type="hash"/>

   <xsd:element name="keyholder" type="keyholder-obj"/>

   <xsd:complexType name="keyholder-obj">
    <xsd:choice>
     <xsd:group ref="principal"/>
     <xsd:element ref="name"/>
    </xsd:choice>
   </xsd:complexType>

   <xsd:element name="k-of-n" type="subj-thresh"/>
```

```
   <xsd:complexType name="subj-thresh">
    <xsd:sequence>
     <xsd:element ref="k-val"/>
     <xsd:element ref="n-val"/>
     <xsd:group ref="subject-obj" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="k-val" type="integer"/>

   <xsd:element name="n-val" type="integer"/>

   <!--                                                                -->
   <!-- Section 4.8: Tags                                              -->
   <!--                                                                -->

   <xsd:element name="tag" type="tag-content"/>

   <xsd:complexType name="tag-content">
    <xsd:choice>
     <xsd:group ref="tag-expression"/>
     <xsd:element ref="tag-null"/>
     <xsd:element ref="tag-star"/>
    </xsd:choice>
   </xsd:complexType>

   <xsd:group name="tag-expression">
    <xsd:choice>
     <xsd:element ref="tag-string"/>
     <xsd:group ref="star-tag"/>
     <xsd:element ref="simple-tag"/>
    </xsd:choice>
   </xsd:group>

   <xsd:element name="tag-null" type="tag-null"/>

   <xsd:complexType name="tag-null"/>

   <xsd:element name="tag-star" type="tag-star"/>

   <xsd:complexType name="tag-star"/>

   <xsd:element name="tag-string" type="byte-string"/>

   <xsd:group name="star-tag">
    <xsd:choice>
     <xsd:element ref="set"/>
     <xsd:element ref="prefix"/>
     <xsd:element ref="range"/>
    </xsd:choice>
   </xsd:group>
```

```
   <xsd:element name="simple-tag" type="simple-tag"/>

   <xsd:complexType name="simple-tag">
    <xsd:sequence>
     <xsd:element ref="type"/>
     <xsd:group ref="tag-expression" minOccurs="0"
                                     maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="type" type="byte-string"/>

   <xsd:element name="set" type="tag-set"/>

   <xsd:complexType name="tag-set">
    <xsd:sequence>
     <xsd:group ref="tag-expression" minOccurs="0"
                                     maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="prefix" type="byte-string"/>

   <xsd:element name="range" type="tag-range"/>

   <xsd:complexType name="tag-range">
    <xsd:sequence>
     <xsd:element name="range-ordering" type="range-ordering"/>
     <xsd:element name="low-lim" type="low-lim" minOccurs="0"/>
     <xsd:element name="up-lim" type="up-lim" minOccurs="0"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="range-ordering" id="range-ordering">
    <xsd:restriction base="xsd:string">
     <xsd:whiteSpace value="collapse"/>
     <xsd:enumeration value="5:alpha"/>
     <xsd:enumeration value="alpha"/>
     <xsd:enumeration value="7:numeric"/>
     <xsd:enumeration value="numeric"/>
     <xsd:enumeration value="4:time"/>
     <xsd:enumeration value="time"/>
     <xsd:enumeration value="6:binary"/>
     <xsd:enumeration value="binary"/>
     <xsd:enumeration value="4:date"/>
     <xsd:enumeration value="date"/>
    </xsd:restriction>
   </xsd:simpleType>
```

```
<xsd:complexType name="limit">
 <xsd:sequence>
  <xsd:element name="limit-type" type="byte-string"/>
  <xsd:element name="limit-value" type="byte-string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="low-lim">
 <xsd:complexContent>
  <xsd:restriction base="limit">
   <xsd:sequence>
    <xsd:element name="limit-type" type="gte"/>
    <xsd:element name="limit-value" type="byte-string"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="gte" id="gte">
 <xsd:restriction base="xsd:string">
  <xsd:whiteSpace value="collapse"/>
  <xsd:enumeration value="1:g"/>
  <xsd:enumeration value="g"/>
  <xsd:enumeration value="2:ge"/>
  <xsd:enumeration value="ge"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="up-lim">
 <xsd:complexContent>
  <xsd:restriction base="limit">
   <xsd:sequence>
    <xsd:element name="limit-type" type="lte"/>
    <xsd:element name="limit-value" type="byte-string"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="lte" id="lte">
 <xsd:restriction base="xsd:string">
  <xsd:whiteSpace value="collapse"/>
  <xsd:enumeration value="1:l"/>
  <xsd:enumeration value="l"/>
  <xsd:enumeration value="2:le"/>
  <xsd:enumeration value="le"/>
 </xsd:restriction>
</xsd:simpleType>
```

```
   <!--                                                              -->
   <!-- Section 4.9: Validity                                        -->
   <!--                                                              -->

   <xsd:element name="validity" type="validity"/>

   <xsd:complexType name="validity">
    <xsd:sequence>
     <xsd:group ref="valid-basic"/>
     <xsd:element ref="online" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:group name="valid-basic">
    <xsd:sequence>
     <xsd:element ref="not-before" minOccurs="0"/>
     <xsd:element ref="not-after" minOccurs="0"/>
    </xsd:sequence>
   </xsd:group>

   <xsd:element name="not-before" type="date"/>

   <xsd:element name="not-after" type="date"/>

   <xsd:simpleType name="date">
    <xsd:restriction base="xsd:string">
     <xsd:whiteSpace value="collapse"/>
     <xsd:pattern value='"[0-9]{4}-(0[1-9]|1[0-2])-
                         (0[1-9]|[1-2][0-9]|3[0-1])_
                         ([0-1][0-9]|2[0-4]):([0-5][0-9]):
                         ([0-5][0-9])"'/>
    </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="online" type="online-test"/>

   <xsd:complexType name="online-test">
    <xsd:sequence>
     <xsd:element ref="online-type"/>
     <xsd:element ref="uris"/>
     <xsd:group ref="principal"/>
     <xsd:element ref="id"/>
     <xsd:group ref="spart" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   </xsd:complexType>
```

```
   <xsd:complexType name="online-test-new-cert">
    <xsd:complexContent>
     <xsd:restriction base="online-test">
      <xsd:sequence>
       <xsd:element name="online-type">
        <xsd:simpleType>
         <xsd:restriction base="xsd:string">
          <xsd:whiteSpace value="collapse"/>
          <xsd:enumeration value="8:new-cert"/>
          <xsd:enumeration value="new-cert"/>
         </xsd:restriction>
        </xsd:simpleType>
       </xsd:element>
       <xsd:element ref="uris"/>
      </xsd:sequence>
     </xsd:restriction>
    </xsd:complexContent>
   </xsd:complexType>

   <xsd:element name="online-type" type="online-type"/>

   <xsd:simpleType name="online-type" id="online-type">
    <xsd:restriction base="xsd:string">
     <xsd:whiteSpace value="collapse"/>
     <xsd:enumeration value="3:crl"/>
     <xsd:enumeration value="crl"/>
     <xsd:enumeration value="5:reval"/>
     <xsd:enumeration value="reval"/>
     <xsd:enumeration value="8:one-time"/>
     <xsd:enumeration value="one-time"/>
    </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="id"       type="byte-string"/>

   <!--                                                                -->
   <!-- Section 5: Name Certificates                                   -->
   <!--                                                                -->

   <xsd:complexType name="name-cert">
    <xsd:complexContent>
     <xsd:restriction base="certificate">
      <xsd:sequence>
       <xsd:element ref="version" minOccurs="0"/>
       <xsd:element ref="display" minOccurs="0"/>
       <xsd:element ref="issuer-name"/>
       <xsd:element ref="subject"/>
       <xsd:element ref="validity" minOccurs="0"/>
       <xsd:element ref="comment" minOccurs="0"/>
      </xsd:sequence>
     </xsd:restriction>
    </xsd:complexContent>
```

```
   </xsd:complexType>

   <xsd:element name="issuer-name" type="issuer-name"/>

   <xsd:complexType name="issuer-name">
    <xsd:sequence>
     <xsd:group ref="principal"/>
     <xsd:element name="name" type="byte-string"/>
    </xsd:sequence>
   </xsd:complexType>

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

   <xsd:complexType name="name">
    <xsd:choice>
     <xsd:group ref="fq-name"/>
     <xsd:group ref="relative-name"/>
    </xsd:choice>
   </xsd:complexType>

   <xsd:group name="fq-name">
    <xsd:sequence>
     <xsd:group ref="principal"/>
     <xsd:element ref="local-name" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:group>

   <xsd:group name="relative-name">
    <xsd:sequence>
     <xsd:element ref="local-name" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:group>

   <xsd:element name="local-name" type="byte-string"/>

   <!--                                                                -->
   <!-- Section 6: ACLs and Sequences                                  -->
   <!--                                                                -->

   <xsd:element name="acl" type="acl"/>

   <xsd:complexType name="acl" id="acl">
    <xsd:sequence>
     <xsd:element ref="version" minOccurs="0"/>
     <xsd:element ref="entry" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
```

```
   <xsd:element name="entry" type="acl-entry"/>

   <xsd:complexType name="acl-entry" id="acl-entry">
    <xsd:sequence>
     <xsd:group ref="subject-obj"/>
     <xsd:element ref="propagate" minOccurs="0"/>
     <xsd:element ref="tag"/>
     <xsd:element ref="validity" minOccurs="0"/>
     <xsd:element ref="comment" minOccurs="0"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="sequence" type="sequence"/>

   <xsd:complexType name="sequence">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
     <xsd:group ref="seq-entry"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:group name="seq-entry">
    <xsd:choice>
     <xsd:element ref="cert"/>
     <xsd:element ref="public-key"/>
     <xsd:element ref="signature"/>
     <xsd:group   ref="op"/>
     <xsd:element ref="reval"/>
     <xsd:element ref="crl"/>
     <xsd:element ref="delta-crl"/>
    </xsd:choice>
   </xsd:group>

   <xsd:element name="do-hash" type="hash-alg-name"/>

   <xsd:element name="do" type="general-op"/>

   <xsd:complexType name="op">
    <xsd:choice>
     <xsd:element ref="do-hash"/>
     <xsd:element ref="do"/>
    </xsd:choice>
   </xsd:complexType>

   <xsd:complexType name="general-op">
    <xsd:sequence>
     <xsd:element ref="operation"/>
     <xsd:group ref="spart" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="operation" type="byte-string"/>
```

```
   <!--                                                                -->
   <!-- Section 7: Online Test Reply Formats                           -->
   <!--                                                                -->

   <xsd:element name="crl" type="crl"/>

   <xsd:complexType name="crl">
    <xsd:sequence>
     <xsd:element ref="version" minOccurs="0"/>
     <xsd:element ref="canceled"/>
     <xsd:group ref="valid-basic"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="canceled" type="hash-list"/>

   <xsd:complexType name="hash-list">
    <xsd:sequence>
     <xsd:element ref="hash" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="delta-crl" type="delta-crl"/>

   <xsd:complexType name="delta-crl">
    <xsd:sequence>
     <xsd:element ref="version" minOccurs="0"/>
     <xsd:element ref="hash"/>
     <xsd:element ref="canceled"/>
     <xsd:group ref="valid-basic"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="reval" type="reval"/>

   <xsd:complexType name="reval">
    <xsd:choice>
     <xsd:sequence>
      <xsd:element ref="version" minOccurs="0"/>
      <xsd:element ref="valid"/>
      <xsd:group ref="valid-basic"/>
     </xsd:sequence>
     <xsd:sequence>
      <xsd:element ref="version" minOccurs="0"/>
      <xsd:element ref="hash"/>
      <xsd:element ref="one-time"/>
     </xsd:sequence>
    </xsd:choice>
   </xsd:complexType>
```

```
   <xsd:element name="valid" type="reval-list"/>

   <xsd:complexType name="reval-list">
    <xsd:sequence>
     <xsd:element ref="hash" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="one-time" type="byte-string"/>

   <!--                                                              -->
   <!-- Integer                                                      -->
   <!--                                                              -->

   <xsd:complexType name="integer" id="integer">
    <xsd:simpleContent>
     <xsd:extension base="byte-string"/>
    </xsd:simpleContent>
   </xsd:complexType>

   </xsd:schema>
```

Appendix C - Full S-Expr XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
   elementFormDefault="qualified" version="1.0">

<xsd:group name="spart">
 <xsd:choice>
  <xsd:element ref="byte-string"/>
  <xsd:element ref="sexpr"/>
 </xsd:choice>
</xsd:group>

<xsd:element name="sexpr" type="sexpr"/>

<xsd:complexType name="sexpr">
 <xsd:sequence>
  <xsd:element name="type" type="byte-string" id="type-sexpr"/>
  <xsd:group ref="spart" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="byte-string" type="byte-string"/>

<xsd:complexType name="byte-string">
 <xsd:simpleContent>
  <xsd:extension base="bytes">
   <xsd:attribute name="display-type" type="display-type"
                  use="optional"/>
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>

<xsd:simpleType name="display-type">
 <xsd:restriction base="xsd:binary">
  <xsd:whiteSpace value="collapse"/>
  <xsd:pattern value="\[(\p{Nd})+:(\p{L}|\p{M}|\p{N}|
                      \p{P}|\p{Z}|\p{S}|\p{C})+\]"/>
  <xsd:pattern value="\[\|(.)+\|\]"/>
  <xsd:pattern value="\[#([0-9]|[A-F]|[a-f])*#\]"/>
  <xsd:pattern value='\["(.)*"\]'/>
  <xsd:pattern value="\[([a-zA-Z\-\./_:\*\+=]
                      [a-zA-Z0-9\-\./_:\*\+=]*)\]"/>
</xsd:restriction>
</xsd:simpleType>
```

```
   <xsd:simpleType name="bytes" id="bytes">
    <xsd:restriction base="xsd:binary">
     <xsd:pattern value="(\p{Nd})+:(\p{L}|\p{M}|\p{N}|
                         \p{P}|\p{Z}|\p{S}|\p{C})+"/>
     <xsd:pattern value="\|(.)+\|"/>
     <xsd:pattern value="#([0-9]|[A-F]|[a-f])*#"/>
     <xsd:pattern value='"(.)*"'/>
     <xsd:pattern value="([a-zA-Z\-\./_:\*\+=]
                         [a-zA-Z0-9\-\./_:\*\+=]*)"/>
    </xsd:restriction>
   </xsd:simpleType>

   </xsd:schema>
```

Appendix D - XSLT Stylesheet for SPKI trans-coding

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
<xsl:output method="text" omit-xml-declaration="yes"
            indent="no" media-type="text/plain"/>
<xsl:strip-space elements="*"/>

<!--                                                                -->
<!-- XSLT for transformation of XML to S-expression SPKI objects -->
<!-- ---------------------------------------------------------- -->
<!-- Based on SPKI-XML Schema in document                       -->
<!--                     <draft-orri-xml-spki-cert-struc-00.txt>  -->
<!-- Note: references to section numbers correspond to section   -->
<!-- numbers in document "XML-SPKI Certificate Structure"        -->

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

<xsl:template match="text()">
    <xsl:variable name="dt">
        <xsl:value-of select="parent::*/@display-type"/>
    </xsl:variable>
    <xsl:text>  </xsl:text>
    <xsl:if test="string-length($dt) != 0">
        <xsl:value-of select="$dt"/>
    </xsl:if>
    <xsl:value-of select="."/>
</xsl:template>

<!--                                                                -->
<!-- Section 3.2: Primitive Objects                                 -->
<!--                                                                -->

<!-- ============================                                  -->
<!-- = Public and Private Keys =                                    -->
<!-- ============================                                  -->

<xsl:template match="public-key">
    <xsl:text>(public-key </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>)</xsl:text>
</xsl:template>
```

```
   <xsl:template match="private-key">
       <xsl:text>(private-key (</xsl:text>
       <xsl:apply-templates/>
       <xsl:text>))</xsl:text>
   </xsl:template>

   <xsl:template match="key-value">
       <xsl:text>(</xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="pub-sig-alg-id">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="any-key-value">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="uris">
       <xsl:text>(uri </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="uri">
       <xsl:apply-templates/>
   </xsl:template>

   <!-- ============================================ -->
   <!-- = RSA Key Values - Public, Private and CRT = -->
   <!-- ============================================ -->

   <xsl:template match="rsa-key-value">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="rsa-priv-key-value">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="rsa-crt-key-value">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="n">
       <xsl:text>(n </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <xsl:template match="e">
       <xsl:text>(e </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="d">
       <xsl:text>(d </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="p">
       <xsl:text>(p </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="q">
       <xsl:text>(q </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="a">
       <xsl:text>(a </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="b">
       <xsl:text>(b </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="c">
       <xsl:text>(c </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!-- ======================================                    -->
   <!-- = DSA Key Values - Public and Private =                    -->
   <!-- ======================================                    -->

   <xsl:template match="dsa-key-value">
       <xsl:apply-templates/>
   </xsl:template>
```

```
   <xsl:template match="dsa-priv-key-value">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="g">
       <xsl:text>(g </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="y">
       <xsl:text>(y </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="x">
       <xsl:text>(x </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!-- <p> already defined for RSA CRT keys                        -->

   <!-- <q> already defined for RSA CRT keys                        -->

   <!-- ==========                                                  -->
   <!-- = Hashes =                                                  -->
   <!-- ==========                                                  -->

   <xsl:template match="hash">
       <xsl:text>(hash </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="hash-alg-name">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="hash-value">
       <xsl:apply-templates/>
   </xsl:template>
```

```
   <!-- ==============                                                -->
   <!-- = Signatures =                                                -->
   <!-- ==============                                                -->

   <xsl:template match="signature">
       <xsl:text>(signature </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="signature-value">
       <xsl:text>(</xsl:text>
           <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="rsa-sig-params">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="dsa-sig-params">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="any-sig-params">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="r">
       <xsl:text>(r </xsl:text>
           <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="s">
       <xsl:text>(s </xsl:text>
           <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!--                                                              -->
   <!-- Section 4: Authorization Certificates                        -->
   <!--                                                              -->

   <xsl:template match="cert">
       <xsl:text>(cert </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <xsl:template match="version">
       <xsl:text>(version </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="display">
       <xsl:text>(diisplay </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="issuer">
       <xsl:text>(issuer </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="issuer-info">
       <xsl:text>(issuer-info </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="subject">
       <xsl:text>(subject </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="subject-info">
       <xsl:text>(subject-info </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="propagate">
       <xsl:text>(propagate) </xsl:text>
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="comment">
       <xsl:text>(comment </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <!-- ====================                                        -->
   <!-- = Subject Objects =                                        -->
   <!-- ====================                                        -->

   <xsl:template match="object-hash">
       <xsl:text>(object-hash </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="keyholder">
       <xsl:text>(keyholder </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="k-of-n">
       <xsl:text>(k-of-n </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="k-val">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="n-val">
       <xsl:apply-templates/>
   </xsl:template>

   <!-- ========                                                   -->
   <!-- = Tags =                                                   -->
   <!-- ========                                                   -->

   <xsl:template match="tag">
       <xsl:text>(tag </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="tag-star">
       <xsl:text>(tag *) </xsl:text>
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="tag-string">
       <xsl:apply-templates/>
   </xsl:template>
```

```
   <xsl:template match="simple-tag">
       <xsl:text>(</xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!-- <tag-null> no rule provided thus nothing output             -->

   <xsl:template match="prefix">
       <xsl:text>(* prefix </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="set">
       <xsl:text>(* set </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="range">
       <xsl:text>(* range </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="range-ordering">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="range-ordering">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="low-lim">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="up-lim">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="limit-type">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="limit-value">
       <xsl:apply-templates/>
   </xsl:template>
```

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

   <!-- ============                                              -->
   <!-- = Validity =                                              -->
   <!-- ============                                              -->

   <xsl:template match="validity">
       <xsl:text>(validity </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="not-before">
       <xsl:text>(not-before </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="not-after">
       <xsl:text>(not-after </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="online">
       <xsl:text>(online </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="online-type">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="id">
       <xsl:text>(id </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!--                                                          -->
   <!-- Section 5: Name Certificates                             -->
   <!--                                                          -->

   <xsl:template match="issuer-name">
       <xsl:text>(issuer (name </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <xsl:template match="name">
       <xsl:text>(name </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="local-name">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="reval">
       <xsl:text>(reval </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="valid">
       <xsl:text>(valid </xsl:text>
           <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <!--                                                              -->
   <!-- Section 6: ACLs and Sequences                                -->
   <!--                                                              -->

   <xsl:template match="acl">
       <xsl:text>(acl </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="entry">
       <xsl:text>(entry </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="sequence">
       <xsl:text>(sequence </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <!--                                                                -->
   <!-- Section 7: Online Test Reply Formats                           -->
   <!--                                                                -->

   <xsl:template match="one-time">
       <xsl:text>(one-time </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="do-hash">
       <xsl:text>(do hash </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="do">
       <xsl:text>(do </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="operation">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="canceled">
       <xsl:text>(canceled </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="crl">
       <xsl:text>(crl </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   <xsl:template match="delta-crl">
       <xsl:text>(delta-crl </xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>
```

```
   <!--                                                              -->
   <!--      XSLT from XML S-expressions to SPKI S-expressions       -->
   <!--     ------------------------------------------------------   -->
   <!-- Based on SPKI-XML Schema in document                         -->
   <!--                     <draft-orri-xml-spki-cert-struc-00.txt>  -->

   <xsl:template match="byte-string">
       <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="sexpr">
       <xsl:text>(</xsl:text>
       <xsl:apply-templates/>
       <xsl:text>)</xsl:text>
   </xsl:template>

   </xsl:stylesheet>
```

Appendix E - Full XML-DTD for SPKI certificates

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT acl (version?, entry*)>
<!ELEMENT any-key-value (sexpr+)>
<!ELEMENT any-sig-params (byte-string | sexpr+)>
<!ELEMENT byte-string (#PCDATA)>
<!ATTLIST byte-string
     display-type CDATA #IMPLIED
>
<!ELEMENT canceled (hash*)>
<!ELEMENT cert ((version?, display?, issuer, issuer-info?, subject,
subject-info?, propagate?, tag, validity?, comment?)|(version?,
display?, issuer-name, subject, validity?, comment?))>
<!ELEMENT comment (#PCDATA)>
<!ATTLIST comment
     display-type CDATA #IMPLIED
>
<!ELEMENT crl (version?, canceled, ((not-before?, not-after?)))>
<!ELEMENT delta-crl (version?, hash, canceled, ((not-before?, not-
after?)))>
<!ELEMENT display (#PCDATA)>
<!ATTLIST display
     display-type CDATA #IMPLIED
>
<!ELEMENT do (operation, ((byte-string | sexpr))+)>
<!ELEMENT do-hash (#PCDATA)>
<!ELEMENT dsa-key-value (p, g?, q, y)>
<!ELEMENT dsa-priv-key-value (p, g?, q, y, x)>
<!ELEMENT dsa-sig-params (r, s)>
<!ELEMENT entry (((((public-key | hash)) | name | object-hash |
keyholder | k-of-n)), propagate?, tag, validity?, comment?)>
<!ELEMENT hash (hash-alg-name, hash-value, uris?)>
<!ELEMENT hash-alg-name (#PCDATA)>
<!ELEMENT hash-value (#PCDATA)>
<!ATTLIST hash-value
     display-type CDATA #IMPLIED
>
<!ELEMENT id (#PCDATA)>
<!ATTLIST id
     display-type CDATA #IMPLIED
>
<!ELEMENT issuer ((public-key | hash))>
<!ELEMENT issuer-info (uri+)>
<!ELEMENT issuer-name (((public-key | hash)), name)>
<!ELEMENT k-of-n (k-val, n-val, ((((public-key | hash)) | name |
object-hash | keyholder | k-of-n))*)>
<!ELEMENT k-val (#PCDATA)>
<!ATTLIST k-val
     display-type CDATA #IMPLIED
>
```

```
<!ELEMENT key-value (pub-sig-alg-id, (rsa-key-value | dsa-key-value
| any-key-value))>
<!ELEMENT keyholder (((public-key | hash)) | name)>
<!ELEMENT local-name (#PCDATA)>
<!ATTLIST local-name
     display-type CDATA #IMPLIED
>
<!ELEMENT n-val (#PCDATA)>
<!ATTLIST n-val
     display-type CDATA #IMPLIED
>
<!ELEMENT name (((((public-key | hash)), local-name+)) | ((local-
name+)))>
<!ELEMENT not-after (#PCDATA)>
<!ELEMENT not-before (#PCDATA)>
<!ELEMENT object-hash (hash-alg-name, hash-value, uris?)>
<!ELEMENT one-time (#PCDATA)>
<!ATTLIST one-time
     display-type CDATA #IMPLIED
>
<!ELEMENT online ((online-type, uris, ((public-key | hash)), id,
((byte-string | sexpr))*) | (online-type, uris))>
<!ELEMENT online-type (#PCDATA)>
<!ELEMENT operation (#PCDATA)>
<!ATTLIST operation
     display-type CDATA #IMPLIED
>
<!ELEMENT prefix (#PCDATA)>
<!ATTLIST prefix
     display-type CDATA #IMPLIED
>
<!ELEMENT private-key (pub-sig-alg-id, (rsa-priv-key-value | rsa-
crt-key-value | dsa-priv-key-value | any-key-value))>
<!ELEMENT propagate EMPTY>
<!ELEMENT pub-sig-alg-id (#PCDATA)>
<!ELEMENT public-key (key-value, uris?)>
<!ELEMENT range (range-ordering, low-lim?, up-lim?)>
<!ELEMENT reval ((version?, valid, ((not-before?, not-after?))) |
(version?, hash, one-time))>
<!ELEMENT rsa-crt-key-value (e?, n, d, p, q, a, b, c)>
<!ELEMENT rsa-key-value (e, n)>
<!ELEMENT rsa-priv-key-value (e?, n, d)>
<!ELEMENT rsa-sig-params (#PCDATA)>
<!ATTLIST rsa-sig-params
     display-type CDATA #IMPLIED
>
<!ELEMENT sequence (((cert | public-key | signature | do-hash | do |
reval | crl | delta-crl)))*>
<!ELEMENT set (((tag-string | ((set | prefix | range)) | simple-
tag))*)>
<!ELEMENT sexpr (type, ((byte-string | sexpr))*)>
<!ELEMENT signature (hash, ((public-key | hash)), signature-value)>
```

```
<!ELEMENT signature-value (pub-sig-alg-id, ((dsa-sig-params | rsa-
sig-params | any-sig-params)))>
<!ELEMENT simple-tag (type, ((tag-string | ((set | prefix | range))
| simple-tag))*)>
<!ELEMENT subject ((((public-key | hash)) | name | object-hash |
keyholder | k-of-n))>
<!ELEMENT subject-info (uri+)>
<!ELEMENT tag (((tag-string | ((set | prefix | range)) | simple-
tag)) | tag-null | tag-star)>
<!ELEMENT tag-null EMPTY>
<!ELEMENT tag-star EMPTY>
<!ELEMENT tag-string (#PCDATA)>
<!ATTLIST tag-string
     display-type CDATA #IMPLIED
>
<!ELEMENT type (#PCDATA)>
<!ATTLIST type
     display-type CDATA #IMPLIED
>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri
     display-type CDATA #IMPLIED
>
<!ELEMENT uris (uri+)>
<!ELEMENT valid (hash+)>
<!ELEMENT validity (((not-before?, not-after?)), online*)>
<!ELEMENT version (#PCDATA)>
<!ATTLIST version
     display-type CDATA #IMPLIED
>
<!ELEMENT p (#PCDATA)>
<!ATTLIST p
     display-type CDATA #IMPLIED
>
<!ELEMENT g (#PCDATA)>
<!ATTLIST g
     display-type CDATA #IMPLIED
>
<!ELEMENT q (#PCDATA)>
<!ATTLIST q
     display-type CDATA #IMPLIED
>
<!ELEMENT y (#PCDATA)>
<!ATTLIST y
     display-type CDATA #IMPLIED
>
<!ELEMENT x (#PCDATA)>
<!ATTLIST x
     display-type CDATA #IMPLIED
>
```

```
<!ELEMENT r (#PCDATA)>
<!ATTLIST r
     display-type CDATA #IMPLIED
>
<!ELEMENT s (#PCDATA)>
<!ATTLIST s
     display-type CDATA #IMPLIED
>
<!ELEMENT range-ordering (#PCDATA)>
<!ELEMENT low-lim (limit-type, limit-value)>
<!ELEMENT up-lim (limit-type, limit-value)>
<!ELEMENT e (#PCDATA)>
<!ATTLIST e
     display-type CDATA #IMPLIED
>
<!ELEMENT n (#PCDATA)>
<!ATTLIST n
     display-type CDATA #IMPLIED
>
<!ELEMENT d (#PCDATA)>
<!ATTLIST d
     display-type CDATA #IMPLIED
>
<!ELEMENT a (#PCDATA)>
<!ATTLIST a
     display-type CDATA #IMPLIED
>
<!ELEMENT b (#PCDATA)>
<!ATTLIST b
     display-type CDATA #IMPLIED
>
<!ELEMENT c (#PCDATA)>
<!ATTLIST c
     display-type CDATA #IMPLIED
>
<!ELEMENT limit-type (#PCDATA)>
<!ELEMENT limit-value (#PCDATA)>
<!ATTLIST limit-value
     display-type CDATA #IMPLIED
>
```

References

[SPKI] C. M. Ellison et al., "Simple Public Key Certificate", <draft-ietf-spki-cert-structure-06.txt>, 26 July 1999, Expired 31 January 2001, Work in Progress. (Available at http://world.std.com/~cme/spki.txt)

[RFC2692] C. M. Ellison, "SPKI Requirements", RFC 2692, September 1999. (Available at ftp://ftp.isi.edu/in-notes/rfc2692.txt)

[RFC2693] C. M. Ellison et al., "SPKI Certificate Theory", RFC 2693, September 1999. (Available at ftp://ftp.isi.edu/in-notes/rfc2693.txt)

[CERTEX] C. M. Ellison et al., "SPKI Examples", <draft-ietf-spki-cert-examples-01.txt>, 10 March 1998, Expired 15 September 1998, Work in Progress. (Available at http://wprld.std.com/~cme/examples.txt)

[SEXP] R. Rivest, code and description of S-expressions, http://theory.lcs.mit.edu/~rivest/sexp.html

[S2X] C. M. Ellison, code and examples of transformation from S-expressions to XML documents, http://world.std.com/~cme/htlm/s2x.zip

[PAAJ] J. Paajarvi, "XML Encoding of SPKI Certificates", <draft-paajarvi-xml-spki-cert-00.txt>, March 2000, Expired September 2000 (Copy available at http://world.std.com/~cme/draft-paajarvi-xml-spki-cert-00.txt)

[XML] T. Bray et al., "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation 6 October 2000, http://www.w3.org/TR/2000/REC-xml-20001006/

[XSL] S. Adler et al., "Extensible Stylesheet Language (XSL) 1.0", W3C Recommendation 15 October 2001, http://www.w3.org/TR/2001/REC-xsl-20011015/

[XSLT] J. Clark, "XSL Transformations (XSLT) Version 1.1", W3C Working Draft 24 August 2001, http://www.w3.org/TR/2001/WD-xslt11-20010824/

[XSIG] M. Bartel et al., "XML-Signature Syntax and Processing", W3C Proposed Recommendation 20 August 2001, http://www.w3.org/TR/2001/PR-xmldsig-core-20010820/

[JAXB] Sun Microsystems, "Java Architecture for XML Binding", Java Community Process JSR-31, http://java.sun.com/xml/jaxb/index.html

Acknowledgments

The work presented in this document is only a small step compared with the huge amount of brainstorming hours behind SPKI. One of our goals in writing this document is to contribute to a wider acceptance of SPKI as standard. But the people to thank for are the numerous minds that have contributed, one way or the other, in the design and definition of SPKI.

We would also want to thank you, the reader of this Internet-Draft, in advance for providing the authors with your valuable feedback, comments, questions, propositions or pieces of advice. This needs to be a collaborative effort and as such, the involvement and ideas of people in SPKI and XML communities, and outside, are of great importance to the authors.

Authors' Addresses

Joan-Maria Mas Ribes
Octalis SA
Av. Albert Einstein, 11-F
B-1348 Louvain-la-Neuve (Belgium)

Phone: +32-10-45.81.99
Mobile: +32-497-45.68.02
Fax: +32-10-45.57.29
E-mail: mas@octalis.com

Xavier Orri Sainz de los Terreros
Octalis SA
Av. Albert Einstein, 11-F
B-1348 Louvain-la-Neuve (Belgium)

Phone: +32-10-45.81.99
Mobile: +32-497-45.68.03
Fax: +32-10-45.57.29
E-mail: orri@octalis.com

Expiration and File Name

This draft expires 31 May 2002.

Its file name is draft-orri-xml-spki-cert-struc-00.txt