CONTENTGUARD™

eXtensible rights Markup Language (XrML) 2.0 Specification
Part II: XrML Core Schema

20 November 2001

Available formats: HTML and PDF. In case of a discrepancy, the HTML is considered definitive.

NOTE: To enable interactive browsing of the XrML schemas and examples, the XrML Specification and its companion Example Use Cases document use an HTML version that leverages the XML access functionality provided by the W3C Xpath recommendation. For this reason, you need to view these HTML documents with a browser that supports that recommendation (for example, Internet Explorer Version 6.0). If your browser does not support this functionality, please view the PDF versions of those documents.

Quick Table of Contents

Full Table of Contents for Part II: XrML Core Schema

5 Technical Reference

This section of the XrML2 specification provides normative technical details regarding the core of the XrML2 design and architecture.

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.

5.1 XrML2 Schema and Schema Structure

The syntax of XrML2 is described and defined using the XML Schema technology defined by the Worldwide Web Consortium (W3C). Significantly more powerful and expressive than DTD technology, the extensive use of XML Schema in XrML2 allows for significant richness and flexibility in its expressiveness and extensibility.

To that end, a principal design goal for XrML2 is to allow for and support a significant amount of extensibility and customizability without the need to make actual changes to the XrML2 standard itself. Indeed, the standard itself makes use of this extensibility internally. XrML2 is split into several parts:

1. A core schema, containing definitions of concepts that are at the heart of the semantics of XrML2, particularly those having to do with evaluation of a trust decision,
2. A standard extension schema, containing definitions of concepts which are generally and broadly useful and applicable to XrML2 usage scenarios, but which aren't necessarily at the heart of XrML2 semantics, and
3. Other, domain-specific extensions, including a content management schema that defines rights-management concepts specifically related to digital works such as books, music, and video.

Each of these XML Schemas is a normative part of the overall XrML2 specification. In particular, the core schema is a normative part of the XrML2 core. Others parties may if they wish define their own extensions to XrML2. This is accomplished using existing, standard XML Schema and XML Namespace mechanisms.

Readers of these schemas should notice that a certain editorial style has for ease of comprehension been uniformly adopted. The XML Schema artifacts found therein fall into two categories: elements and types. The names of each have a different stylistic treatment: the names of types are in mixed case, with an initial capital letter, while the names of elements are in mixed case but with an initial lower case letter. For example, `Grant` is the name of a type, while `grant` is the name of an element. This stylistic convention has also been used in this specification: for example, a passage herein which mentions a `Grant` is using the word in a technical sense to refer to the notion of `Grant` as an XML Schema type.

5.2 Architectural Details of the XrML2 Core

At the heart of XrML2 is the XrML2 Core Schema. The elements and types defined therein define the core structural and validation semantics that comprise the essential essence of the specification. It is expected that every XrML2 validation processor will be aware of the semantics embodied in this core. That is not to say that each and every such processor need to implement and fully support all of the functionality herein described; rather, it indicates that such processors must be conscious of all the semantics defined therein that logically affect those core features they indeed do choose to support. This is also true for XrML2 extensions that these processors intend to process.

5.2.1 License

The single most important concept in XrML2 is that of the **`License`**. A `License` is conceptually a container of **`Grant`**s, each one of which conveys to a particular **`Principal`** the sanction to exercise some identified **`Right`** against some identified **`Resource`**, possibly subject to the need for some **`Condition`** to be first fulfilled. A `License` is also a container of **`GrantGroup`**s, each of which is in turn an eventual container of `Grant`s.

Schema Representation of the `License` Type

```
-<xsd:complexType name="License">
 -<xsd:choice>
  -<xsd:sequence>
   -<xsd:element name="title" type="r:LinguisticString" minOccurs="0" maxOccurs="unbounded">
    </xsd:element>
   -<xsd:element name="inventory" type="r:Inventory" minOccurs="0">
    </xsd:element>
```

```
    -<xsd:choice minOccurs="0" maxOccurs="unbounded">
       <xsd:element ref="r:grant"/>
       <xsd:element ref="r:grantGroup"/>
     </xsd:choice>
     <xsd:element ref="r:issuer" minOccurs="0" maxOccurs="unbounded"/>
    -<xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded">
     </xsd:any>
   </xsd:sequence>
  -<xsd:element name="encryptedLicense" type="r:EncryptedContent">
   </xsd:element>
 </xsd:choice>
 <xsd:attribute name="licenseId" type="xsd:anyURI" use="optional"/>
</xsd:complexType>
```

A License may be digitally signed by the party who issues it, signifying that the License issuer **authorizes** certain Grants and GrantGroups. This semantic notion of whether or not a Grant or GrantGroup has been authorized is an important one. A Grant or GrantGroup which has not been authorized conveys no authorization, it merely exists as an XML element. Unless otherwise indicated by this specification, Grants or GrantGroups which may physically appear in a License are not to be considered authorized

Syntactically, multiple issuers may sign a given License; however no additional semantic is associated with their collective signing. The semantics are, rather, as if they had each independently signed their own copy of the License. Therefore, one can unambiguously speak of *the* issuer of a given License.

5.2.1.1 License/title

Each of the zero or more title elements in a License provides a descriptive phrase about the License that is intended for human consumption in user interfaces and the like. Automated processors MUST NOT interpret semantically the contents of such title elements.

5.2.1.2 License/grant and License/grantGroup

The grant and grantGroup elements contained in a License are the means by which authorization policies are conveyed in the XrML2 architecture. The presence of an issuer's signature on a license indicates that the issuer indeed authorizes the Grants and GrantGroups which are immediate children thereof.

Each grant or grantGroup element which is an immediate child of a license exists independently within a license: no collective semantic (having to do with their particular ordering or otherwise) is intrinsically associated with the presence of two or more of them within a certain one license (though there may be syntactic issues; see license parts).

See below in this specification for an elaboration of the semantics of Grant and GrantGroup.

5.2.1.3 License/issuer

Each issuer element in a License contains two pieces of information:

- a set of issuer-specific details about the circumstances under which he issues the License, and
- a digital signature for the License.

The optional issuer-specific details are found in the License/issuer/details element, which is of type IssuerDetails. These details optionally include any of the following information:

1. the specific date and time at which this issuer claims to have effected his issuance of the License.
2. an interval of time before and after which the issuer does not intend his issuance of the License to be effective.
3. an indication of the mechanism or mechanisms by which the issuer of the License will, if he later revokes it, post notice of such revocation. When checking for revocation, XrML2 processing systems may choose to use *any* one of the identified mechanisms: that is, they are all considered equally authoritative as to the revocation status of the issuance of the License.

Let *g* be any Grant or GrantGroup which is an immediate child of a License *l*, and let *i* be the Issuer element of *l*. If the element *i*/details exists and if *i*/details contains a ValidityInterval *v*, then let the Grant or GrantGroup *g'* be defined to be that Grant or GrantGroup which is formed from a copy of *g* by replacing therein the (possibly absent) element *g'*/condition with an AllConditions condition containing both *v* and (the possibly absent) *g*/condition. Then *g'* is defined to be *directly authorized* by the presence of the signature of the issuer on the License *l* (and *g* itself is not authorized). If instead no such ValidityInterval *v* exists, then, likewise, the Grant or GrantGroup *g* is defined to be *directly authorized* by the presence of the signature of the issuer on the License.

The digital signature made by each issuer of a License is manifest in an XML element of name Signature as defined by the XML-Signature Syntax and Processing standard of the W3C. However, when used within XrML2, some of the general freedoms and flexibilities permitted within that design are profiled and constrained. Specifically, with the aim of simplifying the determination of exactly which pieces of the License have and have not been actually signed by a given issuer, the Signature/SignedInfo/Reference elements are restricted in how they may refer to pieces of the License. In concept, the restriction is that of the information in a License a signature may only reference

a. the whole License less its issuer elements, together with
b. the issuance details paired with this Signature

but not any other piecemeal subparts of the License (the Signature may still, if it wishes, reference items external to the License

though such use is beyond the scope of this specification). Concretely, when an issuer wishes to reference pieces of the License, to do so it MUST use a Signature/SignedInfo/Reference element *r* such that the following is true:

1. the attribute *r*/@URI MUST be omitted
2. the element *r*/Transforms MUST contain exactly one child Transform element *t*, where
   a. *t* MUST be empty
   b. the attribute *t*/@Algorithm MUST contain the value http://www.xrml.org/schema/2001/11/xrml2core#license

The transform algorithm so indicated is known as the *XrML2 License Transform Algorithm*.

A Transform element *t* indicating the use of the XrML2 License Transform Algorithm emits as output the most immediate ancestor of *t* that is of type License or a derivation thereof but with any element descendants of that License which occupy (perhaps through type derivation) the slot defined by the Issuer child of the License wholly removed, except for that Issuer that contains *t*, which is kept, removing its Signature child instead.

It is RECOMMENDED that Signatures created by issuers of XrML2 Licenses indicate the use of the Exclusive XML Canonicalization algorithm.

Moreover, as a general note of good digital signature hygiene, it is RECOMMENDED that XrML2 Licenses explicitly (re)declare no higher up the XML element tree than at the License level any XML Namespaces that are used anywhere throughout the License. That is, a License should be a self-contained unit with respect to XML Namespace declarations, not relying on any such declarations to be imported from their surrounding XML context. This hygienic practice greatly facilitates the ability to manipulate Licenses as a self-contained XML unit within XrML2 processing systems.

5.2.1.4 License/inventory

As is described later, XrML2 provides a syntactic mechanism for reducing redundancy and verbosity in Licenses. This syntactic macro-like mechanism can be used throughout a License, so long as there is in a given License only one definition to each LicensePart. Such definitions can lie, for example, inside of grants or other semantically important structures. However, it is sometimes useful and convenient to be able to provide a definition of a part of a License without at the definition site necessarily associating any particular semantic with the part. The inventory element provides a means for doing this.

The inventory element of a License is a simple container of LicenseParts. The presence of such parts in the inventory container does not provide any semantic at all. The parts simply exist as syntactic structures within the inventory. Usefully and usually, parts in the inventory will have LicensePart/@licensePartId attributes so that they can be referenced from elsewhere in the License.

5.2.1.5 License/(any)

Using the wildcard construct from XML Schema, a License provides an extensibility hook within which License issuers may place additional content as they find appropriate and convenient. This can be useful for conveying information which is peripherally related to, for example, authentication and authorization, but is not part of the XrML2 core infrastructure. Such content will of necessity be referenced by the Signature of the License, and so can be considered as being attested to by the License's issuer; indeed, it is the inclusion of this data in the signature which is likely the most important reason for contemplating the use of this facility.

It should, however, be carefully understood that not all processors of XrML2 Licenses will understand the semantics intended by any particular use of this extensibility hook. That is, there is no means by which the License issuer can *force* such semantics to be adhered to by all processors that may wish to interpret this License. Rather, the issuer must content with the fact that processors of the License may choose wholly at their own discretion to completely ignore any such content that might be present herein.

5.2.1.6 License/encryptedLicense

A mechanism is provided by which the contents of a License may be encrypted and so hidden from view from inappropriate parties. This mechanism makes straightforward use of the XML Encryption Syntax and Processing standard.

Specifically, the XML content model of a License is a choice between a sequence containing the elements previously described in this section and an encryptedLicense element. encryptedLicense represents the encryption of the contents (but not the attributes) of the License element. See the type EncryptedContent for a more detailed discussion of the decryption process.

5.2.2 License Parts

Many of the types defined in XrML2 are, in the XML Schema sense, derivations of the type LicensePart, including Grants, Resources, and Rights, just to name a few.

Schema Representation of the LicensePart Type

```
-<xsd:complexType name="LicensePart" abstract="true">
   <xsd:attribute name="licensePartId" type="r:LicensePartId" use="optional"/>
   <xsd:attribute name="licensePartIdRef" type="r:LicensePartId" use="optional"/>
   <xsd:attribute name="varRef" type="r:VariableName" use="optional"/>
 </xsd:complexType>
```

The role of LicensePart is twofold:

1. `LicensePart`, through its `licensePartId` and `licensePartIdRef` attributes, which are both of type `licensePartId`, defines a macro-like purely syntactic mechanism by which fragments of XML which must logically be present in several places within a `License` may avoid being literally written out multiple times.
2. In contrast, `LicensePart`, through its `varRef` attribute, defines a *semantically* important mechanism. As is later described herein, XrML2 defines a pattern-matching mechanism which may be used, for example, to denote sets of `principal`s that a `grant` might apply to or sets of `grant`s that might be validly issued by an authorized authority. Such patterns logically describe sets of entities. When a pattern is applied to a concrete situation, a matching process occurs, resulting in a single entity that matches that pattern. It is useful to be able to, elsewhere in a `License`, talk about the entity that might match a given pattern when such matching process later occurs.

The matching process and its relationship to variables is somewhat involved, and a detailed discussion is provided later in this specification.

The macro-like facility of `licensePartId` and `licensePartIdRef`, on the other hand, is quite straightforward. Use of the `licensePartId` and `licensePartIdRef` attributes MUST adhere to the following constraints (Note: in the remainder of this section the term 'a `LicensePart`' should be taken to mean 'an element whose type is `LicensePart` or a derivation thereof'):

1. On any given `LicensePart` at most one of the attributes `licensePartId` and `licensePartIdRef` may appear. That is, it is illegal for both attributes to be present on one `LicensePart`.
2. For a given `licensePartId` value *v*, there may be at most one `LicensePart` in a given `License`which contains a `licensePartId` attribute with the value *v*.
3. If a `LicensePart` *p* contains a `licensePartIdRef` attribute, then it MUST have empty content. As a corollary, therefore, it is required that all types which are derivations of `LicensePart` SHOULD allow their content to be empty (for otherwise they cannot usefully be used within the `LicensePart` infrastructure).
4. If a `LicensePart` *p* contains a `licensePartIdRef` attribute with a certain value *v*, then there must exist some (other) `LicensePart` *q* in the same `License` as *p* which has a `licensePartId` attribute with value *v* (and, per (2), there cannot be two such *q*s). It is further required that the expanded element name of *p* exactly match that of *q*. Moreover, it is required that *q* not be an ancestor of *p* (or, per (3), a descendant of *p*).

If a `LicensePart` *p* contains a `licensePartIdRef` attribute with a certain value *v*, and *q* is the `LicensePart` in the same `License` as *p* which has a `licensePartId` attribute with value *v,* then the semantics of the `License` containing *p* and *q* are as if:

a. *p* were removed from the `License` and replaced with a copy *q'* of the element *q*,
b. the `licensePartId` attribute were removed from *q'* and all of its descendants,
c. any "preserved" attributes that may be present on *q'* were removed therefrom, and
d. any "preserved" attributes that may be present on *p* were copied and added to *q'*.

where here a "preserved" attribute is any of the following:

1. any attribute of type ID
2. any attribute for which 'id' is the LocalPart of its qualified name

(It is the intent of the last of these points to allow for the useful definition of other identification systems on license parts beyond the document-global ID-typed identifiers.)

With the exception of signature verification, `licensePartId` macro expansion MUST be carried out before the other `License` processing steps defined by this specification. In particular, it is carried out before the evaluation of variable references is made.

5.2.3 Equality of XML Elements

XrML2 defines a formal notion by which two arbitrary XML elements can be compared and said to be "*equal*" or not. This notion is used extensively and heavily in the design in such places, for example, as determining whether a `Grant` in a particular `License` actually contains a particular `Right` which is attempting to be exercised. In order to determine this, the `Right` being exercised must be compared in a precise and technical manner against the `Right` in the `Grant`. Perhaps surprisingly, no existing notion of equality appears defined on XML elements. Accordingly, we define one here as follows.

Let *x* be a document subset (any single XML element is, in particular, a document subset). Define *c(x)* to be the result of:

1. removing any `licensePartId` attributes from *x* and its descendants, and then
2. applying Exclusive XML Canonicalization to *x* where the InclusiveNamespacePrefix parameter is a list which contains those namespace prefixes which are visibly utilized within *x* but for which the XML namespace prefix declaration which is in-scope at the point of visible usage lies outside of *x* (that is, lies on some ancestor of *x* within the XML document of which *x* is a part), and then
3. removing from the output thereof any occurrences of the xsi:schemaLocation and xsi:noNamespaceSchemaLocation attributes (as these are only hints and not authoritative, especially in security-related systems such as XrML2 processors), and then
4. removing from the output thereof any white space which is physically outside any one or more of the following:
    a. an XML start-tag
    b. an XML end-tag
    c. an XML empty-element tag
    d. an XML processing instruction
    e. an XML comment
    f. the content of an XML element which has (either explicitly or by implication from its schema) an xml:space attribute with value "preserve" and whose effect has not been overridden with another instance of the xml:space attribute.
    g. the content of an XML element whose type according to its schema is any simple type.
    h. the non-subelement content of an XML element whose content type according to its schema is mixed

The intended effect of the last of these steps is to remove white space in situations where it is not of semantic significance to XML schema, such as between end-tags and start-tags of element content.

Let *p* and *q* be document subsets. Then *p* is said to be *equal* to *q* if and only if the octet sequences *c(p)* and *c(q)* are identical.

It is important to understand that the approach by which the specification of the equality of XML elements in this section is described and document is by no means intended to be the best or most efficient manner in which the algorithm can in fact be *implemented.* It is, rather, merely the most succinct and straightforward exposition that the authors of this specification found to communicate the essential details of the algorithm.

5.2.4 Patterns

Within XrML2, it is quite useful and important at times to be able to write in XML formal expressions that semantically denote particular sets of XML instance elements. To give but one example, a `License` that provides to a `Principal` the authorization that is analogous to that held by a "Certificate Authority" in X.509 parlance needs to be able to precisely specify and carefully indicate exactly which set of `Grant`s the principal is authorized to `issue`. XrML2 has a rich architecture of "patterns" designed to address this and similar needs.

5.2.4.1 XmlPatternAbstract

All formal patterns in XrML2 have types which derive from the type `XmlPatternAbstract`. As such, this type forms the root of a type hierarchy of various flavors of patterns suitable for different pattern matching requirements. The corresponding element `xmlPatternAbstract`, which is of this type, usefully forms the head of a substitution group of all possible patterns.

Schema Representation of the `XmlPatternAbstract` Type

```
-<xsd:complexType name="XmlPatternAbstract">
  -<xsd:complexContent>
     <xsd:extension base="r:Resource"/>
   </xsd:complexContent>
 </xsd:complexType>
```

5.2.4.2 XmlExpression

`XmlExpression` provides a means by which patterns written in formal expression languages defined outside of XrML2 can be straightforwardly incorporated herein. The particular expression language used is indicated by the `lang` attribute, which is a URI.

Schema Representation of the `XmlExpression` Type

```
-<xsd:complexType name="XmlExpression" mixed="true">
  -<xsd:complexContent mixed="true">
    -<xsd:extension base="r:XmlPatternAbstract">
       <xsd:attribute name="lang" type="xsd:anyURI" default="http://www.w3.org/TR/1999/REC-xpath-19991116"/>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

The default value for the `lang` attribute is http://www.w3.org/TR/1999/REC-xpath-19991116, which indicates that the contents of the `XmlExpression` contains a string which is an XPath expression. If the expression contained in that string is not of XPath type boolean, then it is to be automatically converted to such as if the function boolean were applied. An element is said to match an `XmlExpression` pattern if the enclosed expression evaluates to true over that element.

All XrML2 processing systems which choose to support the use of any form of XrML2 patterns at all MUST support the use of the http://www.w3.org/TR/1999/REC-xpath-19991116 expression language in `XmlExpression` elements.

5.2.4.3 The 'match-exact' XPath function

All XrML2 processing systems which support the XPath expression language MUST include an additional `match-exact` function in the library that is used to evaluate XPath expressions. This function, used to match regular expressions, is modeled after work carried out in the XPath2 design effort.

The syntax of this function is

```
match-exact (string $srcval, string $regexp) ==> boolean
```

This function returns a boolean which is true if the regular expression that is the value of `$regexp` matches the entirety of the value of `$srcval` and is false otherwise. The regular expression in the value of `$regexp` uses the syntax of regular expressions specified in Appendix F of [XML Schema Part 2: Datatypes]. Comparisons of characters and character strings are performed in the context of the collation sequence specified by the Unicode Collation Algorithm, though it should be noted that not all regular expressions are semantically sensitive to this collation.

The XPath specification defines that the names of XPath library functions are namespace-qualified. To that end, the match-exact function is defined to reside in the XrML2 core namespace: `http://www.xrml.org/schema/2001/11/xrml2core`.

5.2.4.4 PrincipalPatternAbstract / RightPatternAbstract / ResourcePatternAbstract / ConditionPatternAbstract

As an alternative to using patterns written in externally-defined expression languages, it is often useful to define new XML types and elements that, in their intrinsic semantic, define some pattern matching algorithm. This can, of course, be done by simply deriving from XmlPatternAbstract; but, if appropriate to a given situation, deriving one of the four types here might be more useful.

Schema Representation of the PrincipalPatternAbstract Type

```
-<xsd:complexType name="PrincipalPatternAbstract" abstract="true">
  -<xsd:complexContent>
     <xsd:extension base="r:XmlPatternAbstract"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Schema Representation of the RightPatternAbstract Type

```
-<xsd:complexType name="RightPatternAbstract" abstract="true">
  -<xsd:complexContent>
     <xsd:extension base="r:XmlPatternAbstract"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Schema Representation of the ResourcePatternAbstract Type

```
-<xsd:complexType name="ResourcePatternAbstract" abstract="true">
  -<xsd:complexContent>
     <xsd:extension base="r:XmlPatternAbstract"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Schema Representation of the ConditionPatternAbstract Type

```
-<xsd:complexType name="ConditionPatternAbstract" abstract="true">
  -<xsd:complexContent>
     <xsd:extension base="r:XmlPatternAbstract"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Patterns which are of types which derive from PrincipalPatternAbstract, RightPatternAbstract, ResourcePatternAbstract, and ConditionPatternAbstract are always evaluated in a context of an entire XML document which (respectively) contains exactly just one Principal, Right, Resource, or Condition. Such known contextual setting may make it possible to more succinctly express and define the semantics of the intended pattern.

5.2.4.5 Everyone

Everyone is a type which is derived from PrincipalPatternAbstract.

Schema Representation of the Everyone Type

```
-<xsd:complexType name="Everyone">
  -<xsd:complexContent>
    -<xsd:extension base="r:PrincipalPatternAbstract">
      -<xsd:sequence minOccurs="0">
         <xsd:element ref="r:resource"/>
         <xsd:element ref="r:trustedIssuer" minOccurs="0"/>
       </xsd:sequence>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

As such, it matches documents which are elements of some subset of the universe of Principals. That subset is defined as those Principals who posses a certain property described within the Everyone element.

More precisely, let *e* be an instance of Everyone, and let *P* be the set of Principals denoted by *e*. If *e*/resource does not exist, then *P* is defined to be the entire universe of Principals. Otherwise, *P* is defined to be the set of those Principals *p* for which the following PrerequisiteRight condition *q* can be shown to be fulfilled with respect to the same tuple of Authorization Algorithm inputs within which *e* is being processed:

1. *q*/principal is equal to *p*
2. *q*/right is equal to the possesProperty element
3. *q*/resource is equal to *e*/resource
4. *q*/trustedIssuer is a copy of *e*/trustedIssuer (if such is present) or is absent (otherwise).

5.2.4.6 PatternFromLicensePart

PatternFromLicensePart is a semantically simple pattern. Each element of this type contains exactly one LicensePart. The pattern is defined to match exactly those elements which are equal to this contained part.

Schema Representation of the PatternFromLicensePart Type

```
-<xsd:complexType name="PatternFromLicensePart">
```

```
 -<xsd:complexContent>
  -<xsd:extension base="r:XmlPatternAbstract">
   -<xsd:sequence>
      <xsd:element ref="r:licensePart"/>
     </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

5.2.4.7 GrantPattern

A GrantPattern is a relatively complex pattern which matches XML elements of type Grant. Let *G* be a GrantPattern, and let *g* be a target Grant against which one wishes to attempt to match *G*.

Schema Representation of the GrantPattern Type

```
-<xsd:complexType name="GrantPattern">
 -<xsd:complexContent>
  -<xsd:extension base="r:ResourcePatternAbstract">
   -<xsd:sequence>
    -<xsd:choice minOccurs="0">
       <xsd:element ref="r:principal"/>
       <xsd:element ref="r:principalPattern"/>
     </xsd:choice>
    -<xsd:choice>
       <xsd:element ref="r:right"/>
       <xsd:element ref="r:rightPattern"/>
     </xsd:choice>
    -<xsd:choice minOccurs="0">
       <xsd:element ref="r:resource"/>
       <xsd:element ref="r:resourcePattern"/>
     </xsd:choice>
    -<xsd:choice minOccurs="0">
       <xsd:element ref="r:condition"/>
       <xsd:element ref="r:conditionPattern"/>
     </xsd:choice>
    -<xsd:element name="wholeGrantExpression" type="r:XmlExpression" minOccurs="0" maxOccurs="unbounded">
     </xsd:element>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

The GrantPattern *G* can contain four separate pieces, each of which provide sub-patterns which are matched (respectively) in the context of the Principal, Right, Resource, and Condition of the target Grant *g*, along with an optional fifth piece which is matched in the context of *g* as a whole. The overall GrantPattern *G* is considered to successfully match against the target Grant *g* if and only if each of the five pieces which may be present in *G* successfully match against their respective context.

The first piece of a GrantPattern, which is optional, contains either a literal Principal, or several patterns for a Principal. If a literal Principal *p* is provided, then the target Grant *g* must contain as its principal an element that is equal to *p*. If patterns for a Principal are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Principal from the target Grant *g*, must successfully match.

The second piece of a GrantPattern, which for technical reasons is not optional, contains either a literal Right, or several patterns for a Right. If a literal Right *r* is provided, then the target Grant *g* must contain as its right an element that is equal to *r*. If patterns for a Right are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Right from the target Grant *g*, must successfully match. Note that although this second piece of a GrantPattern is required, a pattern of the form

```
<rightPattern/>
```

can be used to match any Right.

The third piece of a GrantPattern, which is optional, contains either a literal Resource *R*, or several patterns for a Resource. If a literal Resource is provided, then the target Grant *g* must contain as its resource an element that is equal to *R*. If patterns for a Resource are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Resource from the target Grant *g*, must successfully match.

The fourth piece of a GrantPattern, which is optional, contains either a literal Condition *c*, or several patterns for a Condition. If a literal Condition is provided, then the target Grant *g* must contain as its Condition an element which is equal to *c*. If patterns for a Condition are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Condition from the target Grant *g*, must successfully match.

The fifth piece of a GrantPattern is also optional. If present, then it is an XmlExpression that, when evaluated in a target context of a new XML document containing the whole target Grant *g*, must successfully match.

5.2.4.8 GrantGroupPattern

Much as GrantPatterns provide a structured way to match against Grants, GrantGroupPatterns provide a structured way to match

against GrantGroups. Let *G* be a GrantGroupPattern, and let *g* be a target GrantGroup against which one wishes to attempt to match *G*. *G* consists of possibly several pieces. The overall GrantGroupPattern *G* is considered to successfully match against the target GrantGroup *g* only if each of the pieces which may be present in *G* successfully match against their respective context.

Schema Representation of the GrantGroupPattern Type

```
-<xsd:complexType name="GrantGroupPattern">
 -<xsd:complexContent>
  -<xsd:extension base="r:ResourcePatternAbstract">
   -<xsd:sequence>
    -<xsd:choice minOccurs="0">
       <xsd:element ref="r:principal"/>
       <xsd:element ref="r:principalPattern"/>
     </xsd:choice>
    -<xsd:choice minOccurs="0">
       <xsd:element ref="r:condition"/>
       <xsd:element ref="r:conditionPattern"/>
     </xsd:choice>
    -<xsd:choice maxOccurs="unbounded">
      -<xsd:choice>
         <xsd:element ref="r:grant"/>
         <xsd:element ref="r:grantPattern"/>
       </xsd:choice>
      -<xsd:choice>
         <xsd:element ref="r:grantGroup"/>
         <xsd:element ref="r:grantGroupPattern"/>
       </xsd:choice>
     </xsd:choice>
    -<xsd:element name="wholeGrantGroupExpression" type="r:XmlExpression" minOccurs="0" maxOccurs="unbounded">
     </xsd:element>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

The first piece of a GrantGroupPattern, which is optional, contains either a literal Principal, or several patterns for a Principal. If a literal Principal *p* is provided, then the target GrantGroup *g* must contain as its principal an element that is equal to *p*. If patterns for a Principal are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Principal from the target GrantGroup *g*, must successfully match.

The second piece of a GrantGroupPattern, which is optional, contains either a literal Condition *c*, or several patterns for a Condition. If a literal Condition is provided, then the target GrantGroup *g* must contain as its condition an element that is equal to *c*. If patterns for a Condition are provided, then each such pattern, when evaluated in a target context of a new XML document containing only the Condition from the target GrantGroup *g*, must successfully match.

The third piece of a GrantGroupPattern consists of a sequence of sub-patterns, each of which is either a literal Grant or pattern for a Grant, or a literal GrantGroup or a pattern for a GrantGroup. Each literal or pattern in this sequence, when evaluated in the context of a new XML document containing only the corresponding Grant or GrantGroup from the sequence thereof at the end of the target GrantGroup *g*, must successfully match. In doing so, sub-patterns which are Grants or GrantGroups are, as one would by now expect, to match elements which are equal to themselves. Further, the sequence of Grants and GrantGroups at the end of *g* can be no longer than that sequence in *G*.

The fourth piece of a GrantGroupPattern is also optional. If present, then it is an XmlExpression that, when evaluated in a target context of a new XML document containing just the whole target GrantGroup *g*, must successfully match.

5.2.5 Variable Definition and Referencing

A particularly powerful and useful construct in Grants and GrantGroups is the definition and use of variables therein. With variables, a single Grant or GrantGroup can be written (and thus can be issued or otherwise authorized) that allows some carefully controlled variation and flexibility in the rights actually conveyed.

5.2.5.1 Variable Definition

Variables are defined using universal quantification as embodied in elements of type ForAll.

Schema Representation of the ForAll Type

```
-<xsd:complexType name="ForAll">
 -<xsd:complexContent>
  -<xsd:extension base="r:LicensePart">
   -<xsd:sequence>
      <xsd:element ref="r:xmlPatternAbstract" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="varName" type="r:VariableName"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Let *f* be an element of type ForAll. The varName attribute of *f* indicates the name of the variable being defined. The elemental contents of *f* are zero or more patterns which determine what the variable *f*/@varName binds to.

If *x* is any XML element, let *d(x)* be a new XML document containing the element *x* as the root. Define *m(x)* to be the boolean function which is true if and only if all of the patterns in *f*, when evaluated in a context of *d*(*x*), successfully matches. Let *B(f)* be that subset of the universe *X* of XML elements such that *m*(*b*) is true for every *b* in *B(f)* and is false for every *b'* in *X* - *B(f)* (note that this implies that if *f* contains no patterns that *B(f)* is the entire universe *X*). The set of *bindings* of the variable *f*/@`varName` is then defined to be the set *B(f)*.

The element *f* has a *scope* within which the variable it defines it may be referenced. Colloquially, that scope is the rest of the parent element in which *f* is contained, less the scope of any other element of type `ForAll` therein which happens to (re)declare the same variable. More precisely, let *N(y)* be that set of XPath nodes selected by the XPath location path:

```
following-sibling::*/descendent-or-self::node()
```

when evaluated with *y* as the contextual XPath node. For an element *z* of type `ForAll`, let *O(z)* be that set of XPath nodes selected by location path:

```
following-sibling::*/decendent-or-self::r:forAll[@r:varName=$fVarName]
```

(where the XML Namespace prefix `r` is bound to the XrML2 core namespace) when evaluated with *z* as the contextual XPath node and `$fVarName` as the value of *z*/@`varName`.

Let *P(f)* be the union over all *w* in *O(f)* of *N(w).* Then the scope of *f* is defined to be *N(f)* less *P(f)*.

The set *S(f)* of the *eligible bindings* of the variable *f*/@`varName`, then, is defined to be that subset of *B(f)* such that *s* in *B(f)* is in *S(f)* if and only if for all elements *t* in the scope of *f* where *t*/@`varRef` equals *f*/@`varName` all of the following hold:

1. Either the expanded element name of *s* must exactly match that of *t* or *s* must be substitutable for *t* using substitution groups (that is, *t* is the head of a substitution group in which *s* resides).
2. If *t* is removed from its document and replaced with a copy of *s*, that document is (still) valid.

5.2.5.2 Variable Referencing

Variables are referenced using the `varRef` attribute of `LicensePart`s. Let *t* be a `LicensePart` or a derivation thereof, and suppose *t*/@`varRef` exists. Then it is required that *t* must be an empty element: from a conceptual perspective, the contents of *t* are determined by the binding of the variable that it references, not from local elements.

Schema Representation of the `LicensePart` Type

```
-<xsd:complexType name="LicensePart" abstract="true">
   <xsd:attribute name="licensePartId" type="r:LicensePartId" use="optional"/>
   <xsd:attribute name="licensePartIdRef" type="r:LicensePartId" use="optional"/>
   <xsd:attribute name="varRef" type="r:VariableName" use="optional"/>
 </xsd:complexType>
```

Moreover, the value in *t*/@`varRef` MUST be the name of some variable *v* whose scope includes *t*.

5.2.6 Grant

A `Grant` is an XML structure that expresses an assertion that some **`Principal`** may exercise some **`Right`** against some **`Resource`**, subject, possibly, to some **`Condition`**. This structure is at the heart of the rights-management and authorization-policy semantics that XrML2 is designed to express.

Schema Representation of the `Grant` Type

```
-<xsd:complexType name="Grant">
 -<xsd:complexContent>
  -<xsd:extension base="r:Resource">
    -<xsd:choice minOccurs="0">
      -<xsd:sequence>
         <xsd:element ref="r:forAll" minOccurs="0" maxOccurs="unbounded"/>
         <xsd:element ref="r:delegationControl" minOccurs="0"/>
         <xsd:element ref="r:principal" minOccurs="0"/>
         <xsd:element ref="r:right"/>
         <xsd:element ref="r:resource" minOccurs="0"/>
         <xsd:element ref="r:condition" minOccurs="0"/>
       </xsd:sequence>
      -<xsd:element name="encryptedGrant" type="r:EncryptedContent">
       </xsd:element>
     </xsd:choice>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

Especially in situations such as content-management scenarios, it is likely to be common practice that one `License` contain several `Grant`s to the same `Principal` pertaining to the same `Resource`, but differing in the specific `Right` being authorized. One `Grant` might authorize a `Play` right, while another might authorize a `Print` right, for example. In other situations, such as those that might mirror the semantics of X.509 certificates, a set of `Grant`s in a `License` might share a `Principal` and a `Right` (perhaps the `PossessProperty` right), but differ in the `Resource` identified. In all such scenarios, it is expected that the syntactic mechanism of

license parts, perhaps together with the use of the `inventory` in the `License`, will be often used to reduce verbosity and to increase the readability of the collective set of `Grant`s.

5.2.6.1 Grant/forAll

At the start of each `Grant` may reside an optional sequence of elements of type `ForAll`. Because of the pattern matching facility therein, this powerful mechanism allows one authorized `Grant` instance to in fact authorize what would otherwise have to be authorized as a set of `Grant`s, a task which may be cumbersome or logistically impossible to actually carry out.

Schema Representation of the `ForAll` Type

```
-<xsd:complexType name="ForAll">
  -<xsd:complexContent>
    -<xsd:extension base="r:LicensePart">
      -<xsd:sequence>
         <xsd:element ref="r:xmlPatternAbstract" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="varName" type="r:VariableName"/>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

The effect of these `ForAll` elements on the semantics of a `Grant` is straightforward. Let *g* be a `Grant` that contains at least one `ForAll` child element, and let *f* be the first such child in *g*. Let *S(f)* be the set of eligible bindings of the variable *f*/@`varName`. For each *s* in *S(f)*, let *g'(s)* be a `Grant` which is equal to a copy of *g* except

1. (the copy of) *f* is not present in *g'(s)*, and
2. throughout the scope of *f* in *g*, all elements containing references to the variable *f*/@`varName` are replaced in *g'(s)* by *s*.

Then, to say that *g* is authorized means that for all such *s*, *g'(s)* is authorized.

Definition: a `Grant` which lacks any children of type `ForAll` (or any constructs that are equivalent thereto, such as an `ExistsRight` condition with a `GrantPattern`) is considered *primitive.*

5.2.6.2 Grant/principal

The element in an instance of a `Grant` that validates against the `principal` element thereof identifies the `Principal` that, under the authority of the `issuer` of the `License`, may exercise the `Right` identified in the `Grant`.

Schema Representation of the `Principal` Type

```
-<xsd:complexType name="Principal">
  -<xsd:complexContent>
     <xsd:extension base="r:Resource"/>
   </xsd:complexContent>
 </xsd:complexType>
```

The element `principal` is itself conceptually abstract; that is, the literal XML element `<principal/>` MUST NOT, except in the form of a variable reference, appear literally in a `Grant`. Elements which are substitutable with `principal` will be used instead. Common examples include the `keyHolder` element and the `allPrincipals` element, though additional useful `Principal`s may be defined in extensions to XrML2.

While the `principal` element of `Grant` is optional within the schema (primarily for the utility this provides to `GrantGroup`s), it is semantically very dangerous to in fact *authorize* a `Grant` which contains no `Principal`. An authorized `Grant` which contains no `Principal` element is considered to be equivalent to an authorized `Grant` that contains an `allPrincipals` `Principal` with zero children, which in turn authorizes the `Grant` to the entire universe of possible `Principal`s.

5.2.6.3 Grant/right

The element in an instance of a `Grant` which validates against the `right` schema component thereof identifies what the issuer of the containing `License` authorizes the indicated `Principal` to actually do.

Schema Representation of the `Right` Type

```
-<xsd:complexType name="Right" abstract="false">
  -<xsd:complexContent>
     <xsd:extension base="r:LicensePart"/>
   </xsd:complexContent>
 </xsd:complexType>
```

The element `right` is itself conceptually abstract; that is, the literal XML element `<right/>` MUST NOT, except in the form of a variable reference, appear in a `Grant`. Rather, designers of particular application domains will define elements which are substitutable with `right`, and these will appear in `Grant`s in place of the `right` element.

5.2.6.4 Grant/resource

Many (but not all) Rights that might be issued are intended to be directed at and authorized against some particular target or Resource. For example, a content-management-related Right which authorizes a Principal to Print must somehow identify exactly what digital work the issuer of the License intends may be printed. In XrML2, this target can be identified as the resource of a Grant. This is accomplished by providing in the Grant instance an element which validates against the resource schema component thereof.

Schema Representation of the Resource Type

```
-<xsd:complexType name="Resource" abstract="false">
  -<xsd:complexContent>
     <xsd:extension base="r:LicensePart"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Like right, the element resource is conceptually abstract, and so, except in the form of a variable reference, MUST NOT appear literally in a Grant. Instead, elements which are substitutable with resource will appear in its place. Such derivations of Resource are defined in schemas which are extensions to the XrML2 core by engineers who understand some appropriate domain-specific Resources against which it is desirous to express authorization policy using an XrML2 License, though some generically useful Resources have been defined, using the same mechanism as would be used by others, as part of the XrML2 core itself.

5.2.6.5 Grant/condition

Principals who authorize Grants often desire the ability to somehow limit or constrain the situations in which the Grant may actually be used. The condition element within a Grant provides a means by which this may be accomplished. If omitted, then no conditions are imposed: the authorized Grant may be used unconditionally. If a condition is present, then the semantic obligations associated with the semantics of that particular condition element must be satisfied with respect to the indicated Grant indicated by the principal therein before it may be used as the basis of an authorization decision.

Schema Representation of the Condition Type

```
-<xsd:complexType name="Condition" abstract="false">
  -<xsd:complexContent>
     <xsd:extension base="r:LicensePart"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Like several other parts of the XrML2 design, the element condition is conceptually abstract, and so, except in the form of a variable reference, MUST NOT appear literally in a Grant. Instead, elements which are substitutable with condition will appear in its place. Such derivations of Condition are defined in schemas which are extensions to the XrML2 core by those wishing to express domain-specific or business-specific conditions that limit in new ways the situations in which an XrML2 Grant may be used to effect an affirmative authorization. That said, several generally useful conditions have been defined within the XrML2 core itself, and in the Standard Extension thereto.

5.2.6.6 Grant/delegationControl

Whenever a Grant is issued, the issuer may optionally indicate in addition that the Grant may be delegated to others. This is accomplished by including in the Grant an element of type DelegationControl; absent such a DelegationControl element, a Grant is not (formally) delegable.

Schema Representation of the DelegationControl Type

```
-<xsd:complexType name="DelegationControl">
  -<xsd:complexContent>
    -<xsd:extension base="r:LicensePart">
      -<xsd:sequence>
        -<xsd:choice>
          -<xsd:element name="maxDepth" type="xsd:nonNegativeInteger">
           </xsd:element>
          -<xsd:element name="infinite">
           </xsd:element>
         </xsd:choice>
        -<xsd:element name="additionalConditionsProhibited" minOccurs="0">
         </xsd:element>
         <xsd:element ref="r:forAll" minOccurs="0" maxOccurs="unbounded"/>
        -<xsd:element name="to" minOccurs="0" maxOccurs="unbounded">
          -<xsd:complexType>
            -<xsd:sequence>
               <xsd:element ref="r:principal"/>
             </xsd:sequence>
           </xsd:complexType>
         </xsd:element>
       </xsd:sequence>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

To say that an authorized Grant *g* is delegable means that the issuer of *g* also authorizes every Grant *g'* where:

1. *g'*/forAll, *g'*/delegationControl, and *g'*/condition are all absent,
2. *g'*/principal is equal to *g*/principal,
3. *g'*/right is equal to the issue element
4. g'/resource is equal to a Grant g'' where
    a. the (possibly empty) sequence of ForAll elements that begins *g* appear as a prefix of the sequence of ForAll elements that begins *g''*
    b. *g''*/delegationControl is compatible with *g*/delegationControl
    c. *g''*/principal is one of the allowable destination principals of *g*/delegationControl
    d. *g''*/right is equal to *g*/right
    e. *g''*/resource is equal to *g*/resource
    f. *g''*/condition is either equal to *g*/condition, or, if *g*/ delegationControl/additionalConditionsProhibited is absent, is equal to the equivalent of an allConditions element which contains at least *g*/condition (if present)

Additional policies which control the circumstances under which *g* is legally delegable are expressed by the semantics embodied in the DelegationControl element; these are explained in detail below. It is to be understood that *g* may be encrypted, and that in such situations the constraints listed here are to be adhered to by the clear-text form of *g*.

5.2.6.7 Grant/encryptedGrant

A mechanism is provided by which the contents of individual Grants may be encrypted and so hidden from view from inappropriate parties. This mechanism makes straightforward use of the XML Encryption Syntax and Processing standard.

Specifically, the XML content model of a Grant is a choice between a sequence containing the elements previously described in this section and an encryptedGrant element. encryptedGrant is of type EncryptedContent and represents the encryption of the contents of the Grant element.

5.2.7 GrantGroup

Within the XrML2 architecture, GrantGroups occupy much the same niche as do their more straightforward cousins, Grants. That is, wherever a Grant may legally appear, it is (usually) the case that a GrantGroup may appear instead, where a GrantPattern may appear, a GrantGroupPattern may take its place, and so on. Indeed, from a point of view of the set of rights actually authorized, the semantics of a GrantGroup can be (and indeed are) specified in terms of the set of rights authorized by a particular set of related Grants. However, from a point of view of pattern matching and inseparability under delegation, issuance, etc., GrantGroups provide additional expressive power not otherwise found in Grants.

Schema Representation of the GrantGroup Type

```
-<xsd:complexType name="GrantGroup">
 -<xsd:complexContent>
  -<xsd:extension base="r:Resource">
   -<xsd:choice minOccurs="0">
    -<xsd:sequence>
       <xsd:element ref="r:forAll" minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element ref="r:delegationControl" minOccurs="0"/>
       <xsd:element ref="r:principal" minOccurs="0"/>
       <xsd:element ref="r:condition" minOccurs="0"/>
      -<xsd:choice maxOccurs="unbounded">
         <xsd:element ref="r:grant"/>
         <xsd:element ref="r:grantGroup"/>
       </xsd:choice>
     </xsd:sequence>
    -<xsd:element name="encryptedGrantGroup" type="r:EncryptedContent">
     </xsd:element>
   </xsd:choice>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

5.2.7.1 GrantGroup/forAll

At the start of each GrantGroup may reside an optional sequence of elements of type ForAll. Because of the pattern matching facility therein, this powerful mechanism allows one authorized GrantGroup instance to in fact authorize what would otherwise have to be authorized as a set of GrantGroups, a task which may be cumbersome or logistically impossible to actually carry out.

The effect of these ForAll elements on the semantics of a GrantGroup is straightforward. Let *g* be a GrantGroup that contains at least one ForAll child element, and let *f* be the first such child in *g*. Let *S(f)* be the set of eligible bindings of the variable *f*/@varName. For each *s* in *S(f)*, let *g'(s)* be a GrantGroup which is equal to a copy of *g* except

1. (the copy of) *f* is not present in *g'(s)*, and
2. throughout the scope of *f* in *g*, all elements containing references to the variable *f*/@varName are replaced in *g'(s)* by *s*.

Then, to say that *g* is authorized means that for all such *s*, *g'(s)* is authorized.

5.2.7.2 GrantGroup/principal and GrantGroup/condition

Having indicated what it means to say that a GrantGroup containing a ForAll element has been authorized, it remains to be specified what it means to say that a GrantGroup which *lacks* any ForAll element has been authorized. Let *g* be such a

GrantGroup lacking a ForAll element, and consider the structure of *g*, which, as is evident in the XrML2 core schema, can be thought of as a sequence containing:

1. an optional DelegationControl element *d*,
2. an optional Principal element *p,*
3. an optional Condition element *c,*
4. one or more contained Grant or GrantGroup elements *g′*.

To say that *g* has been authorized, then, means the following:

1. Consider each such *g′* in *g* where *g′* is a Grant. Let *p′* and *c′* be (respectively) the (possibly absent) principal and (possibly absent) condition contained in *g′*. Let *g″* be a Grant which is equal to *g′* except that

    a. within *g″*, *p′* is replaced by an element equivalent to an allPrincipals element *p″* which in turn contains
        i. *p* (if present)
        ii. *p′* (if present)

    b. within *g″*, *c′* is replaced by an element equivalent to an allConditions element *c″* which in turn contains
        i. *c* (if present)
        ii. *c′* (if present)

    Then to say that the GrantGroup *g* is authorized means that the Grant *g′* is authorized.

2. Similarly, consider each such *g′* in *g* where *g′* is a GrantGroup. Let *p′* and *c′* be (respectively) the (possibly absent) principal and (possibly absent) condition contained in *g′*. Let *g″* be a Grant Group which is equal to *g′* except that

    a. within *g″*, *p′* is replaced by an element equivalent to an allPrincipals element *p″* which in turn contains
        i. *p* (if present)
        ii. *p′* (if present)
    b. within *g″*, *c′* is replaced by an element equivalent to an allConditions element *c″* which in turn contains
        i. *c* (if present)
        ii. *c′* (if present)

    Then to say that the GrantGroup *g* is authorized means that the GrantGroup *g′* is authorized.

The set of authorized Grants which is related to the authorized GrantGroup *g* by means of exhaustive recursive application of Rules (1) and (2) is known as the set of *descendent* Grants of *g*.

A GrantGroup which is an immediate child of a License is considered authorized by the Signature of the issuer of the License.

5.2.7.3 GrantGroup/delegationControl

Whenever a GrantGroup is issued, the issuer may optionally indicate in addition that the GrantGroup may be delegated to others. This is accomplished by including in the GrantGroup an element of type DelegationControl; absent such a DelegationControl element, a GrantGroup is not (formally) delegable.

To say that an authorized GrantGroup *g* is delegable means that the issuer of *g* also authorizes every Grant *g′* where:

1. *g′*/forAll, *g′*/ delegationControl, and *g′*/condition are all absent,
2. *g′*/principal is equal to *g*/ principal,
3. *g′*/right is equal to the issue element
4. *g′*/resource is equal to a GrantGroup *g″* where
    a. the (possibly empty) sequence of ForAll elements that begins *g* appear as a prefix of the sequence of ForAll elements that begins *g″*
    b. *g″*/delegationControl is compatible with *g*/delegationControl,
    c. *g″*/principal is one of the allowable destination principals of *g*/delegationControl
    d. *g″*/condition is either equal to *g*/condition , or, if *g*/condition / additionalConditionsProhibited is absent, is equal to the equivalent of an allConditions element which contains at least *g*/condition (if present)
    e. the Grants and GrantGroups contained as immediate children of *g″* are copies of those contained as immediate children of *g*.

Additional policies which control the circumstances under which *g* is legally delegable are expressed by the semantics embodied in the DelegationControl element; these are explained in detail below. It is to be understood that *g* may be encrypted, and in that in such situations the constraints listed in this section are to be adhered to by the clear-text form of *g*.

5.2.7.4 GrantGroup/encryptedGrantGroup

A mechanism is provided by which the contents of a GrantGroup may be encrypted and so hidden from view from inappropriate parties. This mechanism makes straightforward use of the XML Encryption Syntax and Processing standard.

Specifically, the XML content model of a GrantGroup is a choice between a sequence containing the elements previously described in this section and an encryptedGrantGroup element. encryptedGrantGroup is of type EncryptedContent and represents the encryption of the contents of the GrantGroup element.

5.2.8 DelegationControl

The use of elements of type `DelegationControl` provides the means by which policies which control and otherwise constrain the delegation of `Grant`s and `GrantGroup`s can be expressed.

Schema Representation of the `DelegationControl` Type

```
-<xsd:complexType name="DelegationControl">
 -<xsd:complexContent>
  -<xsd:extension base="r:LicensePart">
   -<xsd:sequence>
    -<xsd:choice>
     -<xsd:element name="maxDepth" type="xsd:nonNegativeInteger">
      </xsd:element>
     -<xsd:element name="infinite">
      </xsd:element>
     </xsd:choice>
    -<xsd:element name="additionalConditionsProhibited" minOccurs="0">
     </xsd:element>
     <xsd:element ref="r:forAll" minOccurs="0" maxOccurs="unbounded"/>
    -<xsd:element name="to" minOccurs="0" maxOccurs="unbounded">
     -<xsd:complexType>
      -<xsd:sequence>
         <xsd:element ref="r:principal"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

Some such policies, namely those regarding constraints on delegated-to `Principal`s and whether additional `Condition`s may be present in delegated `Grant`s and `GrantGroup`s, were described previously herein. Other policies may be defined in types which are derived from the type `DelegationControl`.

5.2.8.1 Allowable Destination Principals

Part of the policy expressed by a `DelegationControl` element *d* is the set of allowable `Principal`s to whom the `Grant` or `GrantGroup` to which *d* is applied may be delegated.

If *d*/to is absent, then the set of *allowable destination principals* of *d* is the universe of all `Principal`s.

Otherwise, at least one *d*/to is present.

Let *z* be a `DelegationControl` that contains at least one child element of type `ForAll`, and let *f* be the first such child in *z*. Let *S(f)* be the set of eligible bindings of the variable *f*/@`varName`. Let $\overline{D}$ be the universe of `DelegationControl` elements. Let *D(z)* be that subset of $\overline{D}$ where *z'* in $\overline{D}$ is in *D(z)* if and only if there exists an *s* in *S(f)* so that *z'* is equal to a copy of *z* except

1. (the copy of) *f* is not present in *z'* and
2. throughout the scope of *f* in *z*, all elements containing references to the variable *f*/@`varName` are replaced in *z'* by *s*.

Now, consider a function *P* defined on the domain $\overline{D}$. For any *z* in $\overline{D}$, let *P(z)* be defined as follows:

1. If *z* has at least one `ForAll` child element, then *P(z)* is the union, over all elements *z'* of the set *D(z)*, of *P(z')*.
2. If *z* does not have at least one `ForAll` child element, then *P(z)* is that set whose members are the `Principal`s found in the `to` elements that are found in *z*.

Then the set of *allowable destination principals* of *d* is that set *P(d)*.

5.2.8.2 Compatibility of DelegationControl Elements

Let *d* and *d'* be `DelegationControl` elements. *d'* is said to be *compatible* with *d* if they are equal except for the following variations:

1. If *d*/`infinite` is present, then *d'*/`maxDepth` may be present (with any nonnegative value)
2. If *d*/`maxDepth` is present, then *d'*/`maxDepth` must be present, and must contain any nonnegative value which is less than the value contained in *d*/`maxDepth`.
3. If *d*/`additionalConditionsProhibited` is absent, then *d'*/`additionalConditionsProhibited` may be present.
4. If *d*/`to` is absent, then any number of *d'*/`to` may be present and identify any `Principal`s.
5. If at least *n* *d*/`to`'s are present where *n*>1, then any *n-1* of them may be omitted in *d'*.
6. If at least one *d*/`to` is present, then *d'*/`to` may contain any `Principal` which is equivalent to an `allPrincipals` `Principal` containing *d*/`to`/`principal` and zero or more arbitrary other `Principal`s.

Notice that "is compatible with" is an antisymmetric and transitive relationship.

5.2.9 EncryptedContent

`EncryptedContent` modifies the semantics of `EncryptedData`, its base type, by simply restricting the use of the `Type` attribute

therein to be the value `http://www.w3.org/2001/04/xmlenc#Content`, which is the type associated with encrypting XML element content. Thus, once decrypted, the plaintext of an element of type `EncryptedContent` is intended to semantically replace the `EncryptedContent` and thus become the content of said element's parent. In doing so, it must of course conform to the schema of the parent as a whole.

Schema Representation of the `EncryptedContent` Type

```
-<xsd:complexType name="EncryptedContent">
  -<xsd:complexContent>
     <xsd:extension base="enc:EncryptedDataType"/>
   </xsd:complexContent>
 </xsd:complexType>
```

5.3 Core Principals

5.3.1 Principal

Within XrML2, instances of the type `Principal` (or a derivation thereof) represent the unique identification of an entity involved in the granting or exercising of rights. In a conceptual sense, they represent the "subject" that is permitted to carry out the action involved in exercising the `Right`.

Schema Representation of the `Principal` Type

```
-<xsd:complexType name="Principal">
  -<xsd:complexContent>
     <xsd:extension base="r:Resource"/>
   </xsd:complexContent>
 </xsd:complexType>
```

The actual type `Principal` is conceptually abstract. That is, it does not indicate how a particular principal is actually identified and authenticated. Rather, this is carried out in types which are derivations of `Principal`. Such derived types may be defined in extensions to XrML2 in order, for example, to provide a means by which `Principal`s who are authenticated using some proprietary logon mechanism may be granted certain `right`s using the XrML2 `License` mechanism. That said, two such derivations are important enough and central enough to be defined within the XrML2 core itself.

5.3.2 The AllPrincipals Principal

Structurally, an `AllPrincipals` `Principal` is a simple container of zero or more other `Principal`s. Semantically, an `AllPrincipals` `Principal` *a* represents the logical conjunct of the `Principal`s represented by all of its children. That is, *a* represents the set of its children *acting together* as one holistic identified entity. For example, if *a* is identified in some `Grant` as that `Principal` which must sign a certain bank loan application, then, conceptually, it is being required that each of the children of *a* act together as co-signers of the loan application.

Schema Representation of the `AllPrincipals` Type

```
-<xsd:complexType name="AllPrincipals">
  -<xsd:complexContent>
    -<xsd:extension base="r:Principal">
      -<xsd:sequence>
         <xsd:element ref="r:principal" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

A corollary of this definition is that an `AllPrincipals` `Principal` which contains zero children identifies the entire universe of possible `Principal`s. Where permitted by the schema in which it is used, such an empty `AllPrincipals` `Principal` is equivalent to said `Principal` in fact being absent.

Note that there is no requirement that a normalization of an `AllPrincipals` `Principal` be carried out. That is, it is perfectly legal for an `AllPrincipals` `Principal` to contain other `AllPrincipals` `Principal`s.

5.3.3 The KeyHolder Principal

Instances of a `KeyHolder` `Principal` represent entities which are identified by their possession of a certain cryptographic key. For example, using a `KeyHolder`, a `Principal` which uses public-key cryptography may be conceptually identified as "that `Principal` which possesses the private key that corresponds to this-here public key." (Indeed, identification of `Principal`s in such a manner is expected to be very common).

Schema Representation of the `KeyHolder` Type

```
-<xsd:complexType name="KeyHolder">
  -<xsd:complexContent>
    -<xsd:extension base="r:Principal">
      -<xsd:sequence minOccurs="0">
         <xsd:element name="info" type="dsig:KeyInfoType"/>
```

```
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

This specification of XrML2 does not itself specify the means by which the key relevant to a KeyHolder is identified. Rather, the info element (which is of type dsig:KeyInfo) within the type KeyHolder element is defined by XrML2 as the mechanism by which such information is conveyed, and the XML-Signature Syntax and Processing specification then specifies the means by which such conveyance is carried out.

5.4 Core Rights

5.4.1 Right

Within XrML2, instances of the type Right (or a derivation thereof) represent a "verb" that a Principal may be authorized to carry out under the authority conveyed by some authorized Grant. Typically, a Right specifies an action (or activity) that a Principal may perform on or using some associated target Resource. The semantic specification of each different particular kind of Right SHOULD indicate which kinds of Resource (if any) may be legally used in authorized Grants containing that Right.

Schema Representation of the Right Type

```
-<xsd:complexType name="Right" abstract="false">
  -<xsd:complexContent>
     <xsd:extension base="r:LicensePart"/>
   </xsd:complexContent>
 </xsd:complexType>
```

The actual type Right is conceptually abstract. That is, the type Right itself does not indicate any actual action or activity that may be carried out. Rather, such actions or activities are to be defined in types which are derivations of Right. Such derived types will commonly be defined in extensions to XrML2, particularly those rights which are germane to a particular application domain. However, several Rights exist which are related to the domain of XrML2 itself, and so are defined within the XrML2 core.

5.4.2 The Issue Right

When an Issue element is used as the right in an authorized Grant *g*, it is required that *g*/resource against which the Right is applied in fact be a Grant or GrantGroup *g'*. The Grant *g* then conveys the authorization for the Principal *g*/ principal to Issue *g'*; that is, it conveys the authorization, under the authority of the issuer of the License *l* within which *g* is authorized, for *g*/principal to Issue other licenses *l'* within which *g'* is authorized.

Schema Representation of the Issue Type

```
-<xsd:complexType name="Issue">
  -<xsd:complexContent>
     <xsd:extension base="r:Right"/>
   </xsd:complexContent>
 </xsd:complexType>
```

Use of the Issue Right is one of the basic mechanisms (along with delegation and trust of a License by some externally specified means) by which the XrML2 Authorization Process chains its processing from one License to another.

Those familiar with the X.509 certificate infrastructure will recognize that, in analogy, the Principal *g*/principal found in an authorized Grant *g* containing the Issue Right can conceptually be considered a "Certificate Authority."

At the instant a License is issued, the Issue right must be held by the issuer of the License with respect to all the Grants and GrantGroups directly authorized therein.

5.4.3 The Revoke Right

The authorized act of exercising the Revoke Right by a Principal *p* effects a retraction of a Signature that was previously issued (either by *p* or by some other Principal from which *p* received appropriate authorization to revoke) and thus accomplishes a withdrawal of any authorization conveyed by that Signature.

Schema Representation of the Revoke Type

```
-<xsd:complexType name="Revoke">
  -<xsd:complexContent>
     <xsd:extension base="r:Right"/>
   </xsd:complexContent>
 </xsd:complexType>
```

There is, of course, commonly a latency, possibly a significant one, between the discovery of an issued Signature by some party wishing to rely on the authorization so conveyed and the subsequent discovery by that party of a later retraction thereof. In the interim, the relying party can and will consider the Signature as valid and binding.

Every issuer of a License, by the act of affixing its Signature thereto, is implicitly and automatically authorized in a freely delegable

manner to subsequently Revoke that Signature, should it choose to do so. By explicit use of the Revoke Right, an issuer may convey that authorization to other Principals of its choosing.

Although the XrML2 core requires that when the Revoke Right is used that the associated Resource explicitly identify the to-be-revoked Signature in question, the core itself does not define a concrete XML data type by which this can be accomplished, instead choosing to leave such definitions to extensions of the core. The XrML2 Standard Extension, though, does define the Resource Revocable which is useful in this role.

At the instant at which a Signature is formally revoked, the Revoke right must be held by the revoking Principal with respect to the Signature being revoked.

5.4.4 The PossessProperty Right

The use of the PossessProperty Right within authorized Grants allows the issuers thereof to straightforwardly express the fact that they authorize the association of property-like characteristics with certain Principals. Put another way, the PossessProperty Right represents the Right for the associated Principal to claim ownership of a particular characteristic, which is listed as the Resource associated with this Right.

Schema Representation of the PossessProperty Type

```
-<xsd:complexType name="PossessProperty">
 -<xsd:complexContent>
    <xsd:extension base="r:Right"/>
  </xsd:complexContent>
 </xsd:complexType>
```

The PossessProperty Right imposes no restriction on the Resource with which it may be used within an authorized Grant other than the fact that such a Resource MUST NOT be omitted. The XrML2 core does not itself define any Resources which are particularly useful for use with the PossessProperty Right. However, several such Resources are defined within the XrML2 Standard Extension; in particular, it defines several Resources which are useful for modeling the authorized binding of names to Principals as is done in the X.509 certificate infrastructure.

Use of the the PossessProperty Right is also very convenient in modeling notions of "group membership" found (among other places) in security systems of traditional operating systems. In this paradigm, in an XrML2 extension one invents a Resource *t* whose associated semantic is "is member of group". Then, straightforwardly, one issues Licenses with authorized Grants that contain the Right possessProperty and the Resource *t* in order to indicate that the associated Principal is in fact a member of the group.

5.4.5 The Obtain Right

When an Obtain element is used as the Right in an authorized Grant *g*, the Resource contained in *g* MUST be present and MUST either be a Grant or a GrantGroup. Let *g'* be that Grant or GrantGroup. Then the semantics conveyed by the authorization of *g* is that the issuer thereof promises that upon request it will in fact issue *g',* subject *only* to the limitation that *g*/principal must first satisfy the (possibly absent) Condition *g*/ condition (the means and manner by which such request to actually issue *g'* is actually carried out is outside the scope of this specification). Thus, the use of the Obtain Right can be conceptualized as an "offer" or "advertisement" by the issuer of the Grant *g* to, for example, "sell" the Grant *g'*.

Schema Representation of the Obtain Type

```
-<xsd:complexType name="Obtain">
 -<xsd:complexContent>
    <xsd:extension base="r:Right"/>
  </xsd:complexContent>
 </xsd:complexType>
```

5.5 Core Resources

5.5.1 Resource

Continuing our grammatical analogy, an instance of type Resource (or a derivation thereof) represents the "direct object" against which the "subject" Principal of a Grant has the Right to perform some "verb." It should be noted that not all XrML2 Rights make use of such target Resources, just as not all verbs require direct objects.

Schema Representation of the Resource Type

```
-<xsd:complexType name="Resource" abstract="false">
 -<xsd:complexContent>
    <xsd:extension base="r:LicensePart"/>
  </xsd:complexContent>
 </xsd:complexType>
```

The actual type Resource is conceptually abstract. That is, the type Resource itself does not indicate any actual object against which a Right may be carried out. Rather, such target objects are to be defined in types which are derivations of Resource. Such derived types will commonly be defined in extensions to XrML2, particularly those Resources which are germane to a particular

application domain. However, several Resources exist which related to the domain of XrML2 itself and so are defined within the XrML2 core

5.5.2 DigitalResource

Use of a DigitalResource Resource in a Grant provides a means by which an arbitrary sequence of digital bits can be identified as being the target object of relevance within the Grant. Specifically, and importantly, such bits are not required to be character strings which conform to the XML specification, but may be arbitrary binary data.

Schema Representation of the DigitalResource Type

```
-<xsd:complexType name="DigitalResource">
 -<xsd:complexContent>
  -<xsd:extension base="r:Resource">
   -<xsd:choice minOccurs="0">
    -<xsd:element name="nonSecureIndirect" type="r:NonSecureReference">
     </xsd:element>
    -<xsd:element name="secureIndirect" type="dsig:ReferenceType">
     </xsd:element>
    -<xsd:element name="binary" type="xsd:base64Binary">
     </xsd:element>
    -<xsd:element name="xml">
     -<xsd:complexType mixed="true">
      -<xsd:sequence>
        <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    -<xsd:any namespace="##other" processContents="lax">
     </xsd:any>
   </xsd:choice>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Conceptually, an instance *d* of DigitalResource defines an algorithm by which a sequence of bits *b* in question is to be *located*. The means by which this is accomplished breaks down in to several cases:

1. The bits *b* are are to be physically present within *d*. There are two sub-cases:
   a. If *b* is a character string which is a sequence of zero or more XML elements, then *b* MAY be represented using the xml element within *d*, which is a simple container of arbitrary XML elements.
   b. Otherwise, *b* SHOULD be encoded in base64 and located within *d* by use of the binary element. Note that there is no requirement that a *b* which may been legally represented using the xml element in fact be represented as such; base64 encoding may equally well be used, even for XML elements.
2. The bits are are to be physically located at some external location outside of *d*. Perhaps, for example, they are located somewhere else within the XML document within which *d* is found, or perhaps at a location on a Web site. There are again two sub-cases:
   a. Though the bits may be external, *d* may still wish to indicate the *exact actual sequence* of bits being referred to. This is accomplished with use of the secureIndirect element.
   b. Otherwise, *d* wishes only to indicate the algorithm used to locate the bits, but is comfortable with the fact that differing actual executions of the algorithm may yield different sequences of bits. This is indicated by the use of the nonSecureIndirect element.
3. The means by which the bits are located is something else which is defined in an extension to XrML2. This is indicated within *d* by the use of an element which validates against the xsd:any element therein.

The secureIndirect element straightforwardly makes use of the cryptographically-secure referencing mechanism designed as part of the XML Signature Syntax and Processing standard, specifically the type ReferenceType defined therein. The documentation of the semantics and processing associated with that type are not described in the present specification but rather are found in the specification of that standard.

The nonSecureIndirect element makes use of an XrML2-defined type NonSecureReference. The structure and attendant semantics of the NonSecureReference type are identical in every way to that of the aforementioned ReferenceType except that

1. NonSecureReference structurally lacks the DigestMethod and DigestValue elements found in ReferenceType, and
2. The processing semantics within ReferenceType that are associated with these two elements (in order to verify that the bits retrieved during the processing of the reference were exactly those expected) are omitted.

5.5.2.1 Authorization of Located Bits

Let *g* be any authorized Grant containing a Resource *d* which is a DigitalResource. Let *b* be the sequence of bits which is the result of any execution of the location algorithm of *d*. Then the Grant *g'* which is identical to *g* except that *d* is replaced by a DigitalResource which contains a child binary element which contains a base64 encoding of *b* is also authorized.

5.6 Core Conditions

5.6.1 Condition

Within XrML2, instances of the type Condition (or a derivation thereof) represent a grammatical "terms & conditions" clause that a

Principal must satisfy before it may take advantage of an authorization conveyed to it in a Grant containing the Condition instance. The semantic specification of each different particular kind of Condition MUST indicate the details of the terms, conditions, and obligations that use of the Condition actually imposes. When these requirements are fulfilled, the Condition is said to be *satisfied*.

Schema Representation of the Condition Type

```
-<xsd:complexType name="Condition" abstract="false">
  -<xsd:complexContent>
     <xsd:extension base="r:LicensePart"/>
   </xsd:complexContent>
 </xsd:complexType>
```

When a particular Condition is used within an authorized Grant, XrML2 processing systems that process the Grant MUST honor the request implied thereby that the terms, conditions, and obligations indicated in the semantic specification of the Condition be satisfied by the Principal indicated in the Grant before the Grant may be used as the basis of an authorization decision. A corollary of this requirement is the observation that should an XrML2 processing system in the course of honoring such a request encounter a Condition defined in some XrML2 extension of which it lacks semantic knowledge, the processing system MUST NOT consider the Condition to be satisfied.

The actual type Condition is conceptually abstract. That is, the type Condition itself does not indicate the imposition of any actual term or condition. Rather, such terms and conditions are to be defined in types which are derivations of Condition. Such derived types will commonly be defined in extensions to XrML2, particularly those Conditions which are germane to a particular application domain. However, several Conditions exist which are related to the domain of XrML2 itself, and so are defined within the XrML2 core.

5.6.2 The AllConditions Condition

Structurally, the AllConditions Condition is a simple container of zero or more other Conditions. Semantically, the AllConditions Condition represents a logical conjunct of the Conditions represented by all of these children. That is, the Conditions imposed by each and every of these children must be satisfied in order for the AllConditions Condition to be satisfied.

Schema Representation of the AllConditions Type

```
-<xsd:complexType name="AllConditions">
  -<xsd:complexContent>
    -<xsd:extension base="r:Condition">
      -<xsd:sequence>
         <xsd:element ref="r:condition" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

A corollary of this definition is that an AllConditions Condition which contains zero children is considered always to be satisfied. It is thus equivalent to the empty AllConditions Condition being absent.

Note that there is no requirement that a normalization of an AllConditions Condition be carried out. That is, it is perfectly legal for an AllConditions Condition to contain other AllConditions Conditions.

5.6.3 The ValidityInterval Condition

A ValidityInterval Condition indicates a contiguous, unbroken interval of time.

Schema Representation of the ValidityInterval Type

```
-<xsd:complexType name="ValidityInterval">
  -<xsd:complexContent>
    -<xsd:extension base="r:Condition">
      -<xsd:sequence>
        -<xsd:element name="notBefore" type="xsd:dateTime" minOccurs="0">
         </xsd:element>
        -<xsd:element name="notAfter" type="xsd:dateTime" minOccurs="0">
         </xsd:element>
       </xsd:sequence>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

The semantics of the Condition expressed is that the interval of the exercise of a Right to which a ValidityInterval is applied must lie wholly within this interval. The delineation of the interval is expressed by the presence, as children of the Condition, of up to two specific fixed time instants:

1. the optional `notBefore` element, of type dateTime, indicates the inclusive instant in time at which the interval begins; if absent, the interval is considered to begin at an instant infinitely distant in the past
2. the optional `notAfter` element, also of type dateTime, indicates the inclusive instant in time at which the interval ends; if

absent, the interval is considered to end at an instant infinitely distant in the future.

5.6.4 The RevocationFreshness Condition

As was discussed previously, issuers of XrML2 Licenses may in a License indicate the means by which they will, should they later decide to revoke their Signature, post notice of such revocation. As a practical matter, many if not most of the mechanisms used for such dissemination of revocation information involve a periodic polling on the part of XrML2 processing systems to determine whether new revocation information is available. With such polling necessarily comes a latency of information dissemination. Use of a RevocationFreshness Condition in a Grant or GrantGroup can place an upper bound on the size of this polling latency whenever the Grant or GrantGroup is used as part of an authorization decision.

Schema Representation of the RevocationFreshness Type

```
-<xsd:complexType name="RevocationFreshness">
 -<xsd:complexContent>
  -<xsd:extension base="r:Condition">
    -<xsd:sequence minOccurs="0">
      -<xsd:choice>
        -<xsd:element name="maxIntervalSinceLastCheck" type="xsd:duration">
         </xsd:element>
        -<xsd:element name="noCheckNecessary">
         </xsd:element>
       </xsd:choice>
     </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

If a RevocationFreshness Condition found in an authorized Grant or GrantGroup *g* contains a maxIntervalSinceLastCheck element, and the length of the duration *d* indicated therein is greater than zero, then in order for the Condition to be satisfied, the length of real, wall-clock time that has elapsed between

1. the last time that the Signature on the License *l* in which *g* was authorized was polled to check for revocation, and
2. the time at which *l* is passed as a relevant input License to the XrML2 Authorization Algorithm

must be less than or equal to *d*. If the length of such duration *d* is zero, then in order for the Condition to be satisfied, a poll to check for revocation must be carried out each and every time *l* is passed as a relevant input License in a non-recursive call to the XrML2 Authorization Algorithm. The length of the duration *d* MUST NOT be less than zero.

A RevocationFreshness Condition containing a noCheckNecessary element is defined to be semantically equivalent to what a RevocationFreshness Condition containing a maxIntervalSinceLastCheck element with an infinite duration would signify, but for the fact that the XML Schema duration data type cannot express such infinite durations of time. This policy is an explicit affirmation that revocation need not ever be explicitly polled, in contrast to an omitted RevocationFreshness condition, which leaves the tolerable polling latency to be determined by other means.

5.6.5 The ExistsRight Condition

Schema Representation of the ExistsRight Type

```
-<xsd:complexType name="ExistsRight">
 -<xsd:complexContent>
  -<xsd:extension base="r:Condition">
    -<xsd:sequence minOccurs="0">
      -<xsd:choice>
         <xsd:element ref="r:grant"/>
         <xsd:element ref="r:grantPattern"/>
         <xsd:element ref="r:grantGroup"/>
         <xsd:element ref="r:grantGroupPattern"/>
       </xsd:choice>
       <xsd:element ref="r:trustedIssuer" minOccurs="0"/>
     </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

5.6.5.1 Some Grants Containing ExistsRight Conditions Are Not Primitive

Let *c* be a Condition of type ExistsRight, and let *g* be a Grant containing *c*. Suppose *c*/grantPattern or *c*/grantGroupPattern exists, and let *e* be this element.

Then, as was previously mentioned, *g* is not primitive.

Define the Grant *g'* as being a copy of *g* except for the transformations defined as follows:

1. an additional new forAll element *f* is inserted at the end of the (possibly empty) sequence of ForAll elements that begins *g'*, where
2. *f*/@varName contains a new variable name which is different from the name of any other variable defined within *g',* and

3. the contents of *f* is the element *e*, and
4. the element *e* within *c* is replaced with an empty (respectively) `grant` or `grantGroup` element which contains a reference to the variable named in *f*/@`varName`.

If *g* is authorized, then *g'* is also authorized.

5.6.5.2 Satisfaction of ExistsRight

Let the functions *P* and *Q*, and the notation *allPrincipals(P)* be as defined in the XrML2 Authorization Algorithm. Let t_0 be the present time.

Let *c* be an `ExistsRight` condition returned from a call to the XrML2 Authorization Algorithm whose inputs were *(p, r, t, v, L, R, C, T).* It follows that either *c*/`grant` or *c*/`grantGroup` exists; let *h* be that element. Then, in order for *c* to be satisfied,

1. If c/`trustedIssuer` exists, it must be established that there exists a time instant *i* prior to *v* and a `Principal` *p'* from those that conform to the policy articulated within the element c/`trustedIssuer` such that *P(p')* is a subset of $Q(h, i, v, L, C, t_0)$.
2. If c/`trustedIssuer` does not exist, it must be established that there exists a time instant *i* prior to *v* for which the call to the XrML2 Authorization Algorithm with inputs:

   ($allPrincipals(Q(h, i, v, L, C, t_0))$, the `issue` element, *h, i, L, R, C, T* union {*h*})

   either

   a. returns ***yes***, or
   b. returns ***maybe*** together with a set C' of `Condition`s, and at least one `Condition` c' in C' can be shown (possibly with the help of C) to have been satisfied during *i* with respect to this issuance.

5.6.6 The PrerequisiteRight Condition

The `PrerequisiteRight` `Condition` is related to the `ExistsRight` `Condition`, but they differ in many respects. While the `ExistsRight` `Condition` deals with determining if certain `Grant`s and `GrantGroup`s are directly and correctly authorized by some `trustedIssuer`, the `PrerequisiteRight` `Condition` deals with determining that (under the authorization of some `trustedIssuer`) a given `Principal` has a given `Right` to a given `Resource` subject to either no `Condition` or a `Condition` that can be shown to be satisfied.

Schema Representation of the `PrerequisiteRight` Type

```
-<xsd:complexType name="PrerequisiteRight">
 -<xsd:complexContent>
  -<xsd:extension base="r:Condition">
   -<xsd:sequence minOccurs="0">
      <xsd:element ref="r:principal" minOccurs="0"/>
      <xsd:element ref="r:right"/>
      <xsd:element ref="r:resource" minOccurs="0"/>
      <xsd:element ref="r:trustedIssuer" minOccurs="0"/>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

5.6.6.1 Satisfaction of PrerequisteRight

Let the functions *P* and *Q*, and the notation *allPrincipals(P)* be as defined in the XrML2 Authorization Algorithm. Let t_0 be the present time.

Let *c* be a `PrerequisiteRight` `Condition` returned from a call to the XrML2 Authorization Algorithm whose inputs were *(p, r, t, v, L, R, C, T).* Then, in order for *c* to be satisfied, it must be shown that there exists some `Grant` or `GrantGroup` *h* such that

1. If c/`trustedIssuer` exists, it must be established that there exists a time instant *i* prior to *v* and a `Principal` *p'* from those that conform to the policy articulated within the element c/`trustedIssuer` such that *P(p')* is a subset of $Q(h, i, v, L, C, t_0)$.
2. If c/`trustedIssuer` does not exist, it must be established that there exists a time instant *i* prior to *v* for which the call to the XrML2 Authorization Algorithm with inputs:

   ($allPrincipals(Q(h, i, v, L, C, t_0))$, the `issue` element, *h, i, L, R, C, T* union {*h*})

   either

   a. returns ***yes***, or
   b. returns ***maybe*** together with a set C' of `Condition`s, and at least one `Condition` c' in C' can be shown (possibly with the help of C) to have been satisfied during *i* with respect to this issuance.
3. There exists a primitive `Grant` *g* such that *g*/`principal` equals *c*/`principal` (or both are absent), *g*/`right` equals *c*/`right`, *g*/`resource` equals *c*/`resource` (or both are absent), the authorization of *h* implies the authorization of *g*, and *g*/`condition` is shown (possibly with the help of *C*) to have been satisfied with respect to the aforesaid algorithm inputs.

5.7 Other Core Types and Elements

5.7.1 TrustedPrincipal

Elements of type `TrustedPrincipal` (or a derivation thereof) indicate a policy by which `Principal`s are identified as having the appropriate and necessary qualifications in order to be trusted for use in certain situations (see, for example, the use of `TrustedPrincipal` in the `ExistsRight` `Condition`).

Schema Representation of the `TrustedPrincipal` Type

```
-<xsd:complexType name="TrustedPrincipal">
  -<xsd:complexContent>
    -<xsd:extension base="r:LicensePart">
      -<xsd:choice minOccurs="0">
         <xsd:element ref="r:principal"/>
        -<xsd:element name="any">
          -<xsd:complexType>
            -<xsd:sequence>
               <xsd:element ref="r:principal" maxOccurs="unbounded"/>
             </xsd:sequence>
           </xsd:complexType>
         </xsd:element>
       </xsd:choice>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

Within `TrustedPrincipal`, this policy is indicated in one of two ways:

1. If the element `TrustedPrincipal`/ `principal` is present, then the set of identified `Principal`s is exactly that one `Principal`.
2. If the element `TrustedPrincipal`/`any` is present, then the set of identified `Principal`s is any of the `Principal`s contained therein.

It is often usefully the case that the `Principal`s within a `TrustedPrincipal` contain references to variables which denote a set of `Principal`s by means of a pattern within a `ForAll` element.

5.7.2 ServiceReference

The term *service* as used in this specification refers to an active body of software, execution of which is distinguished from that of *client* software which wishes to make use of it.

Schema Representation of the `ServiceReference` Type

```
-<xsd:complexType name="ServiceReference">
  -<xsd:complexContent>
    -<xsd:extension base="r:Resource">
      -<xsd:sequence minOccurs="0">
        -<xsd:choice>
          -<xsd:sequence>
            -<xsd:element name="wsdl" type="r:DigitalResource">
             </xsd:element>
            -<xsd:element name="service" type="xsd:NCName">
             </xsd:element>
            -<xsd:element name="portType" type="xsd:NCName" minOccurs="0">
             </xsd:element>
           </xsd:sequence>
          -<xsd:sequence>
            -<xsd:element name="kind">
              -<xsd:complexType>
                -<xsd:sequence>
                  -<xsd:element name="wsdl" type="r:DigitalResource">
                   </xsd:element>
                  -<xsd:element name="binding" type="xsd:NCName">
                   </xsd:element>
                 </xsd:sequence>
               </xsd:complexType>
             </xsd:element>
            -<xsd:element name="address">
              -<xsd:complexType>
                -<xsd:sequence>
                   <xsd:any namespace="##other" processContents="lax"/>
                 </xsd:sequence>
               </xsd:complexType>
             </xsd:element>
           </xsd:sequence>
          -<xsd:element name="uddi" type="r:UddiServiceIdentifier">
           </xsd:element>
          -<xsd:any namespace="##other" processContents="lax">
           </xsd:any>
         </xsd:choice>
        -<xsd:element name="serviceParameters" minOccurs="0">
          -<xsd:complexType>
            -<xsd:sequence minOccurs="0" maxOccurs="unbounded">
              -<xsd:element name="datum">
```

```
            -<xsd:complexType>
              -<xsd:sequence>
                 <xsd:any namespace="##any" processContents="lax"/>
               </xsd:sequence>
             </xsd:complexType>
           </xsd:element>
          -<xsd:element name="transforms" type="dsig:TransformsType" minOccurs="0">
           </xsd:element>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
   </xsd:sequence>
 </xsd:extension>
</xsd:complexContent>
</xsd:complexType>
```

t is the role of an instance of ServiceReference to indicate the location and the means and manner by which a client is to interact with a specific service. Specifically, a ServiceReference instance does the following:

1. Identifies the location or *address* at which the service is found.
2. Identifies a greater or lesser amount of *metadata* about the semantics of the service and the rules that must be adhered to by a client that interacts with it
3. Optionally specifies a set of concrete *parameters* that are to be provided when a client interacts with the service by dereferencing this particular ServiceReference. These parameters provide a means by which a service might at run time distinguish between its uses from different XrML2 contexts.

XrML2 does not itself invent significant new infrastructure for describing services; rather, it draws on the considerable work being done in this area by others. Specifically, there are two architected technologies by which the location and metadata information of a ServiceReference may be provided (using an xsd:any element, ServiceReference provides for other technologies that may also be used):

1. WSDL, the Web Services Definition Language, and
2. UDDI, the Universal Description, Discovery, and Integration directory infrastructure.

5.7.2.1 WSDL

Briefly (see the WSDL specification for details), a WSDL language expression occurs in an WSDL definitions element. This element is a container of *services*. Each WSDL service is a container of *ports*, each of which denotes a different aspect or sub-service of the service. Each port is associated with a particular abstract *portType* and also indicates a *binding* of that abstract portType to concrete message formats and protocol details. Within a service, all ports that share a portType are to be considered as semantically equivalent by clients. The linkage between ports and portTypes, ports and bindings, etc. is by name. Indeed, structurally, a definitions element is a physical container for services, bindings, portTypes, and (perhaps considerably) other metadata.

XrML2 allows for two stylistically different approaches to using WSDL. Let *r* be a ServiceReference element.

In the first approach, the *r*/`wsdl` element, which is of type DigitalResource, is used to locate a WSDL definitions element *d*. The *r*/`service` element then indicates the name of a particular WSDL service *s* that is defined in *d*. Optionally, the *r*/`portType` element disambiguates which subset of the possibly many ports of *s* the ServiceReference *r* intends to refer to.

In the second approach, the *r*/`kind`/`wsdl` element again locates a definitions element *d*. The element r/`kind`/`binding` then indicates the name of a WSDL binding which is defined within *d*. The binding in turn indicates the abstract `portType` of the service, together with a mapping to concrete message formats and protocol details. Remaining to be specified is a concrete endpoint or address at which the software executing the service may be found. This is indicated in the element *r*/`address`.

Which of these two approaches is appropriate depends on the operational details and logistics of the context in which a given DigitalResource might actually be used. In some situations one will be more useful, in different situations, the other will be.

5.7.2.2 UDDI

UDDI defines (see the UDDI specification for details) the notion of a *`registry`* as a particular service replicated over a set of nodes. Each `registry` is a database or directory containing possibly many *`businessEntities`*. Each `businessEntity` contains possibly many *`businessServices`*. Each `businessService` has possibly several *`bindingTemplates`*, each of which may contain explicit endpoint information and also other arbitrary metadata about the `businessService` using data in the form of what are known as *`tModels`*. Several components of the UDDI data model have associated primary keys by which their instances are independently retrievable. These include `businessEntities`, `businessServices`, `bindingTemplates`, and `tModels`.

To uniquely identify a service which is specified using UDDI, one need only identify the `registry`, then identify the primary key of the `businessService` in question within that registry.

Let *r* be a ServiceReference. If *r*/`uddi` is present, then the service referenced by *r* is specified using UDDI.

If *r*/`uddi`/`registry` is omitted, then the registry in question is the Universal Business Registry (the UBR, which is publicly accessible on the Internet and operated by a consortium of companies including IBM, Microsoft, and others). If *r*/`uddi`/`registry` is present, then the value found therein indicates the name (note: not the *location*) of the registry to be used. This name is assumed to be drawn from a list of names of non-UBR UDDI registries known to and useful within the context of usage of the License in which *r* is found.

r/`uddi`/`serviceKey` indicates the primary key of the `businessService` in question within the identified registry. Depending on which

version of UDDI is used, one of two different types of primary key is appropriate. UDDI v1 and v2 use XOpen DCE UUIDs as keys; UDDI v3 and above allows for the use of URIs. Each is available as a choice under *r*/`uddi`/`serviceKey`.

5.7.2.3 Parameters

Let *r* be a `ServiceReference`. Then *r* may contain an ordered sequence of contextual parameters which, per the metadata associated with the service, may be necessary in order to successfully interact with the it. Such parameters may be specified using the sequence contained within the *r*/`serviceParameters` element.

r/`serviceParameters` contains a sequence of pairs of elements. Each pair contains a `datum` element and an optional `transforms` element. Each such `datum` element defines a raw parameter for the service. This raw parameter may be processed to form an actual parameter for the service by applying the sequence of transformations to the raw parameter optionally indicated in the accompanying `transforms` element (if no such transformations are indicated, then the actual parameter is the same as the raw parameter). The specification of the sequence of transformations to be carried out makes use of a mechanism designed as part of the XML Signature Syntax and Processing standard, specifically the type `TransformsType` defined therein. The documentation of the semantics and processing associated with that type are found in the specification of that standard, but the following modifications are made thereto:

1. The input to the first `Transform` is a raw parameter, manifest as an XPath node-set containing the one raw parameter element (that is, the child of the `datum` element) in-place in the context of its XML document (thus navigation from the parameter node to elsewhere in the XML document containing the parameter is feasible).
2. The output of the last `Transform` is the corresponding actual parameter.

`ServiceReference` parameter transformation is defined to take place after all `LicensePart` and variable reference processing has occurred. The use of the parameter transformation facility is in fact particularly convenient in order to be able to discern and communicate to the service the result of such other processing actions.

The actual interpretation, detailed processing, and passing to the service of the sequence of actual parameters is necessarily service-specific, and is thus not defined here.

5.7.3 LicenseGroup

Instances of the type `LicenseGroup` are simple and straightforward containers of `License`s. No inherent semantic is conveyed by the presence of two particular `License`s within the same `LicenseGroup`. This type exists due merely to the observation that it is often handy and convenient to be able to use such a container in XML instances and schemas. No use of it is made in the remainder of XrML2.

Schema Representation of the `LicenseGroup` Type

```
-<xsd:complexType name="LicenseGroup">
 -<xsd:sequence>
    <xsd:element ref="r:license" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
```

5.8 The XrML2 Authorization Algorithm

At the heart of any implementation of software which makes an authorization decision using XrML2 `License`s in the decision making process is a central algorithm (the 'Authorization Algorithm') which answers the question "Is such-and-such a `Principal` authorized to exercise such-and-such a `Right` against such-and-such a `Resource`?" In order for XrML2 to be pragmatically useful, certain details of that algorithm need to be standardized across all such implementations. It is the purpose of this section to document such details.

It is important to understand that the Authorization Algorithm works in terms of potentialities. That is, it colloquially answers the question "If the principal wanted to ..., could he?". A question which is quite a different one is "The principal is about to ...; can he?" The former question addresses a potentiality that might later come to pass; the latter question carries with it the implication that the `Principal` has already committed itself to try to carry out the act. This difference in perspective may be subtle, but could have important implications as to the details of how and when the evaluation of certain kinds of `Condition`s are carried out.

It is also important to understand that the algorithm operates on clear-text `License`s, `Grant`s, and `GrantGroup`s. Encrypted forms of these are to be treated as if they were actually their clear-text equivalent.

Finally, it is important to understand that the approach by which the specification of the Authorization Algorithm in this section is described and document by no means intended to be the best or most efficient manner in which the algorithm can in fact be *implemented*. It is, rather, merely the most succinct and straightforward exposition that the authors of this specification found to communicate the essential details of the algorithm.

5.8.1 Input to the Authorization Algorithm

The Authorization Algorithm takes a number of pieces of information as input:

1. A `Principal` ***p***, which is the identity of the entity whose authorization to perform an act is being called into question,
2. A `Right` ***r***, which embodies the semantics of the action to be performed or otherwise carried out,
3. An (optional) `Resource` ***t***, which is the target of the action ***r*** being carried out by ***p***,
4. An interval ***v*** of time during which the execution of ***r*** by ***p*** is considered to take place. This may either be an instantaneous

point in time, or may be a contiguous, unbroken interval of time.
5. A set ***L*** of relevant `License`s. The algorithm will attempt to find authorized `Grant`s and `GrantGroup`s within these `License`s that it can use to establish a basis for an affirmative authorization decision,
6. An additional set ***R*** of "root" `Grant`s that are considered by the algorithm to be authorized under the authority of an omnipotent issuer. These are authorized `Grant`s that are to be trusted by some decision making process that is outside of the scope of XrML2 itself.
7. A (possibly empty) set ***C*** of other appropriate contextual information. This contextual information is not processed or manipulated directly by the core Authorization Algorithm, and the details of such information are not herein specified, but its existence is established in order to clearly allow for the provision of additional contextual information necessary to evaluate authorization decisions based on `Principal`s, `Right`s, `Resource`s, and `Condition`s that might be defined in extensions to XrML2.
8. A set ***T*** of traversed `Grant`s and `GrantGroup`s. This set is used to ensure that the Authorization Algorithm terminates. The `Grant`s and `GrantGroup`s in this set have already been traversed by parent recursive calls to the algorithm. As such, their authorization should be considered not provable in child calls, and no further recursion should be carried out in an attempt to prove their authorization.

This input can be considered as a eight-tuple:

(***p***, ***r***, ***t***, ***v***, ***L***, ***R***, ***C***, ***T***)

5.8.2 Output of the Authorization Algorithm

The output of the Authorization Algorithm is either:

1. The result ***no***, indicating that the Algorithm could not establish that the `Principal` had the indicated authorization, or
2. Either
   a. the result ***yes***, indicating that the Algorithm established that the `Principal` unequivocally has the indicated authorization, or
   b. the result ***maybe*** together with a non-empty set of alternative `Condition`s, indicating that the `Principal` has the indicated authorization provided that at least one of the indicated alternative `Condition`s is satisfied.

It is important to notice that the core Authorization Algorithm herein described does not itself consider whether or not any particular `Condition` has in fact been satisfied with respect to the input authorization request; such processing and evaluation is (from a specification perspective at least) left to higher level algorithms of the XrML2 processing system which consumes the output of the Authorization Algorithm. That said, in the chaining steps of the Authorization Algorithm, where recursive use of the algorithm is made, such evaluation of `Condition`s output from the recursion is indeed carried out; however, it is there done with respect to rights involved in the authority to `issue` XrML2 `License`s in the input set ***L*** (a `Right` which has been exercised), not the input `Right` ***r*** being requested by the input `Principal` ***p*** (a `Right` that may only *potentially* be exercised).

5.8.4 Execution of the Authorization Algorithm

The execution of the Authorization Algorithm proceeds as follows. We begin with the definition of several important concepts.

Let

- $\overline{\underline{\boldsymbol{P}}}$ be the universe of `Principal`s,
- $\overline{\underline{\boldsymbol{C}}}$ be the universe of `Condition`s,
- $\overline{\underline{\boldsymbol{G}}}$ be the universe of `Grant`s,
- $\overline{\underline{\boldsymbol{GG}}}$ be the universe of `GrantGroup`s,
- $\overline{\underline{\boldsymbol{I}}}$ be the universe of time instants,
- $\overline{\underline{\boldsymbol{V}}}$ be the universe of time intervals
- $\overline{\underline{\boldsymbol{L}}}$ be the universe of `License`s
- $\overline{\underline{\boldsymbol{CC}}}$ be the universe of Authorization Algorithm input contexts

Let $\overline{\underline{\boldsymbol{H}}}$ be the union of $\overline{\underline{\boldsymbol{G}}}$ and $\overline{\underline{\boldsymbol{GG}}}$.

Consider a function *P* defined on the domain $\overline{\underline{\boldsymbol{P}}}$ union $\overline{\underline{\boldsymbol{H}}}$. For any *p* in $\overline{\underline{\boldsymbol{P}}}$, let *P(p)* be defined as follows:

1. If *p* is of type `AllPrincipals`, then *P(p)* is the union, over all children *p'* of *p,* of *P(p')*.
2. If *p* is not of type `AllPrincipals`, then *P(p)* is the one-element set containing *p*.

Colloquially, *P(p)* is the set of `Principal`s obtained by collapsing any `AllPrincipals` elements in *p*. Similarly, for any *h* in $\overline{\underline{\boldsymbol{H}}}$, let *P(h)* be defined as follows:

1. If *h*/`principal` is absent, *P(h)* is the empty set
2. If *h*/`principal` is not absent, *P(h)* is defined to be *P(h*/`principal`*)*

Colloquially, *P(h)* is the set of `Principal`s, acting together, to whom a `Grant` or `GrantGroup` is issued.

Let *S* be any finite subset of $\overline{\underline{\boldsymbol{P}}}$. Then, let the notation *allPrincipals(S)* denote an `allPrincipals` element which contains as children exactly the elements of *S*.

Let *PG* be be that subset of $\overline{\underline{\boldsymbol{G}}}$ where *g* in $\overline{\underline{\boldsymbol{G}}}$ is in *PG* if and only if *g* is primitive. Let *EPG* be that subset of *PG* where *g* in *PG* is in *EPG* if and only if:

1. *P(g)* is a subset of ***P(p)***,
2. *g*/`right` is equal to **r**,
3. either *g*/`resource` is equal to ***t*** or both are absent

EPG can be considered the set of "eligible" primitive `Grant`s.

Let *LH* be that subset of $\overline{\underline{\boldsymbol{H}}}$ where *h* in $\overline{\underline{\boldsymbol{H}}}$ is in *LH* if and only if there exists a `License` *l* in ***L*** in which *h* is directly authorized. Let *ULH* be that subset of *LH* where *h* in *LH* is in *ULH* if and only if *h* is not in ***T***. *ULH* can be considered the set of "usable licensed `Grant`s and `GrantGroup`s."

We define a notion for the set of `Principal`s that have directly authorized a `Grant` or `GrantGroup` prior to a certain time instant. Let *Q* be the function with domain $\overline{\underline{\boldsymbol{H}}}$ x $\underline{\boldsymbol{I}}$ x $\overline{\underline{\boldsymbol{V}}}$ x $\overline{\underline{\boldsymbol{L}}}$ x $\overline{\underline{\boldsymbol{CC}}}$ x $\underline{\boldsymbol{I}}$ and range in $\underline{\boldsymbol{P}}$ which defined as follows: For any *h* in $\overline{\underline{\boldsymbol{H}}}$, *i* and *t* in $\underline{\boldsymbol{I}}$, *v* in $\overline{\underline{\boldsymbol{V}}}$, *L* a set of `License`s, and *C* an authorization context, if *p* is in $\underline{\boldsymbol{P}}$, then *p* is in *Q(h, i, v, L, C, t)* if and only if there exists a `License` *l* in *L* such that

1. *h* is directly authorized within *l*
2. *l* is issued by *p,* and such issuance is not known to have been revoked as of the minimum of times *t* and the end of *v*
3. *p* can be demonstrated to have issued *l* prior to *i* by means of:
    a. a trusted (according to the context *C*) counter-signature for *p*'s signature on *l* attesting to this fact,
    b. *i* being greater than the time *t*
    c. any other method using *C*

We consider the subset of the usable licensed `Grant`s and `GrantGroup`s which are in fact authorized. Let t_0 be the time at which the execution of the Authorization Algorithm occurs. Let *AULH* be that subset of *ULH* where *h* in *ULH* is in *AULH* if and only if there exists a *i* in $\underline{\boldsymbol{I}}$ prior to the start of ***v*** for which a recursive call to the Authorization Algorithm with inputs

(*allPrincipals(Q(h, i,* ***v***, ***L***, ***C,*** t_0*))*, the `issue` element, *h*, *i*, ***L***, ***R***, ***C***, ***T*** union {h})

either

1. returns ***yes***, or
2. return ***maybe*** together with a set *C'* of `Condition`s, and at least one `Condition` *c* in *C'* can be shown (possibly with the help of ***C***) to have been satisfied during *i* with respect to this issuance.

Let *AEPG* be the set of affirmatively authorized eligible primitive `Grant`s defined as follows: *g* in *EPG* is in *AEPG* if and only if there exists an *h* in (*AULH* union ***R***) such that the authorization of *h* implies the authorization of *g*.

If *AEPG* is empty, the Authorization Algorithm returns ***no***.

If there exists a *g* in *AEPG* such that *g*/`condition` is equivalent to an `AllConditions` `Condition` that has no children, then the Authorization Algorithm returns ***yes***.

Otherwise, the Authorization Algorithm returns ***maybe*** together with a set *C* of `Condition`s, where *C* is that subset of $\overline{\underline{\boldsymbol{C}}}$ where *c* in $\overline{\underline{\boldsymbol{C}}}$ is in *C* if and only if there exists a `Grant` *g* in *AEPG* with *g*/`condition` equal to *c*.

This concludes the specification of the Authorization Algorithm.

Go to Part III: Standard Extension