Internet Engineering Task Force (IETF)
Request for Comments: 6025
Category: Informational
ISSN: 2070-1721

C. Wallace
Cygnacom Solutions
C. Gardiner
BBN Technologies

October 2010

ASN.1 Translation

Abstract

Abstract Syntax Notation One (ASN.1) is widely used throughout the IETF Security Area and has been for many years. Some specifications were written using a now deprecated version of ASN.1 and some were written using the current version of ASN.1. Not all ASN.1 compilers support both older and current syntax. This document is intended to provide guidance to specification authors and to implementers converting ASN.1 modules from one version of ASN.1 to another version without causing changes to the "bits on the wire". This document does not provide a comprehensive tutorial of any version of ASN.1. Instead, it addresses ASN.1 features that are used in IETF Security Area specifications with a focus on items that vary with the ASN.1 version.

Table of Contents

1. Introduction

This document is intended to serve as a tutorial for converting ASN.1 modules written using [CCITT.X208.1988] to [CCITT.X680.2002], or vice versa. Preparation of this document was motivated by [RFC5911] and [RFC5912], which provide updated ASN.1 modules for a number of RFCs.

The intent of this specification is to assist with translation of ASN.1 from one version to another without resulting in any changes to the encoded results when using the Basic Encoding Rules or Distinguished Encoding Rules [CCITT.X209.1988] [CCITT.X690.2002]. Other encoding rules were not considered.

Transforming a new ASN.1 module to an older ASN.1 module can be performed in a fairly mechanical manner; much of the transformation consists of deleting new constructs. Transforming an older ASN.1 module to a newer ASN.1 module can also be done fairly mechanically, if one does not wish to move many of the constraints that are contained in the prose into the ASN.1 module. If the constraints are to be added, then the conversion can be a complex process.

1.1. Terminology

This document addresses two different versions of ASN.1. The old (1988) version was defined in a single document (X.208) and the newer (1998, 2002) version is defined in a series of documents (X.680, X.681, X.682, and X.683). For convenience, the series of documents is henceforth referred to as X.68x. Specific documents from the series are referenced by name where appropriate.

2. ASN.1 Design Elements

When translating an ASN.1 module from X.208 syntax to X.68x syntax, or vice versa, many definitions do not require or benefit from change. Review of the original ASN.1 modules updated by [RFC5911] and [RFC5912] and the revised modules included in those documents indicates that most changes can be sorted into one of a few categories. This section describes these categories.

2.1. Open Types

Protocols often feature flexible designs that enable other (later) specifications to define the syntax and semantics of some features. For example, [RFC5280] includes the definition of the Extension structure. There are many instances of extensions defined in other specifications. Several mechanisms to accommodate this practice are available in X.208, X.68x, or both, as described below.

2.1.1. ANY DEFINED BY

X.208 defines the ANY DEFINED BY production for specifying open types. This typically appears in a SEQUENCE along with an OBJECT IDENTIFIER that indicates the type of object that is encoded. The ContentInfo structure, shown below from [RFC5652], uses ANY DEFINED BY along with an OBJECT IDENTIFIER field to identify and convey arbitrary types of data. Each content type to be wrapped in a ContentInfo is assigned a unique OBJECT IDENTIFIER, such as the id-signedData shown below. However, X.208 does not provide a formal means for establishing a relationship between a type and the type identifier. Any associations are done in the comments of the module and/or the text of the associated document.

```
-- from RFC 5652
ContentInfo ::= SEQUENCE {
    contentType ContentType,
    content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER

id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
   us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }
```

ANY DEFINED BY may also appear using an INTEGER to indicate the type of object that is encoded, as shown in the following example from [RFC5280].

```
-- from RFC 5280
ExtensionAttribute ::=  SEQUENCE {
    extension-attribute-type [0] IMPLICIT INTEGER
        (0..ub-extension-attributes),
    extension-attribute-value [1]
        ANY DEFINED BY extension-attribute-type }
```

Though the usage of ANY DEFINED BY was deprecated in 1994, it appears in some active specifications. The AttributeValue definition in [RFC5280] uses ANY with a DEFINED BY comment to bind the value to a type identifier field.

```
-- from RFC 5280
AttributeTypeAndValue ::= SEQUENCE {
    type     AttributeType,
    value    AttributeValue }

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY -- DEFINED BY AttributeType
```

2.1.2. OCTET STRINGs and BIT STRINGs

Both X.208 and X.68x allow open types to be implemented using OCTET STRINGs and BIT STRINGs as containers. The definitions of Extension and SubjectPublicKeyInfo in [RFC5280] demonstrate the usage of OCTET STRING and BIT STRING, respectively, to convey information that is further defined using ASN.1.

```
-- from RFC 5280
Extension  ::=  SEQUENCE  {
    extnID      OBJECT IDENTIFIER,
    critical    BOOLEAN DEFAULT FALSE,
    extnValue   OCTET STRING
    -- contains the DER encoding of an ASN.1 value
    -- corresponding to the extension type identified
    -- by extnID
}

SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }
```

In both cases, the prose of the specification describes that the adjacent OBJECT IDENTIFIER value indicates the type of structure within the value of the primitive OCTET STRING or BIT STRING type. For example, where an extnID field contains the value id-ce-basicConstraints, the extnValue field contains an encoded BasicConstraints as the value of the OCTET STRING, as shown in the dump of an encoded extension below.

```
Tag Length      Value
30   15:         SEQUENCE {
06    3:           OBJECT IDENTIFIER basicConstraints (2 5 29 19)
01    1:           BOOLEAN TRUE
04    5:           OCTET STRING, encapsulates {
30    3:               SEQUENCE {
01    1:                 BOOLEAN TRUE
       :                 }
       :               }
       :           }
```

2.1.3. Information Object Classes

Information object classes are defined in [CCITT.X681.2002]. Object classes allow protocol designers to relate pieces of data that are in some way associated. In the most generic of terms, an Information Object class can be thought of as a database schema, with Information Object Sets being instances of the databases.

Unlike type definitions, object classes with the same structure are not equivalent. Thus, if you have:

```
FOO ::= TYPE-IDENTIFIER

BAR ::= TYPE-IDENTIFIER
```

FOO and BAR are not interchangeable.

TYPE-IDENTIFIER is one of the predefined information object classes in Annex A of [CCITT.X681.2002]. This provides for a simple mapping from an OBJECT IDENTIFIER to a data type. The tag UNIQUE on &id means that this value may appear only once in an Information Object Set; however, multiple objects can be defined with the same &id value.

[RFC5911] uses the TYPE-IDENTIFIER construction to update the definition of ContentInfo, as shown below.

```
-- TYPE-IDENTIFIER definition from X.681
TYPE-IDENTIFIER ::= CLASS
{
    &id OBJECT IDENTIFIER UNIQUE,
    &Type
}
WITH SYNTAX {&Type IDENTIFIED BY &id}

-- from updated RFC 5652 module in [RFC5911]
CONTENT-TYPE ::= TYPE-IDENTIFIER
ContentType ::= CONTENT-TYPE.&id

ContentInfo ::= SEQUENCE {
    contentType        CONTENT-TYPE.
                   &id({ContentSet}),
    content            [0] EXPLICIT CONTENT-TYPE.
                   &Type({ContentSet}{@contentType})}

ContentSet CONTENT-TYPE ::= {
    --  Define the set of content types to be recognized.
    ct-Data | ct-SignedData | ct-EncryptedData | ct-EnvelopedData |
    ct-AuthenticatedData | ct-DigestedData, ... }

-- other CONTENT-TYPE instances not shown for brevity
ct-SignedData CONTENT-TYPE ::=
     { SignedData IDENTIFIED BY id-signedData}
```

This example illustrates the following:

- Definition of an information object class: TYPE-IDENTIFIER and CONTENT-TYPE are information object classes.
- Definition of an information object, or an instance of an information object class: ct-SignedData is an information object.
- Definition of an information object set: ContentSet is an information object set.
- Usage of an information object: The definition of ContentInfo uses information from the CONTENT-TYPE information object class.
- Defining constraints using an object set: Both the contentType and content fields are constrained by ContentSet.

As noted above, TYPE-IDENTIFIER simply associates an OBJECT IDENTIFIER with an arbitrary data type. CONTENT-TYPE is a TYPE-IDENTIFIER. The WITH SYNTAX component allows for a more natural language expression of information object definitions.

ct-SignedData is the name of an information object that associated the identifier id-signedData with the data type SignedData. It is an instance of the CONTENT-TYPE information object class. The &Type field is assigned the value SignedData, and the &id field is assigned the value id-signedData. The example above uses the syntax provided by the WITH SYNTAX component of the TYPE-IDENTIFIER definition. An equivalent definition that does not use the provided syntax is as follows:

```
ct-SignedData CONTENT-TYPE ::=
{
    &id id-signedData,
    &Type SignedData
}
```

ContentSet is the name of a set of information objects derived from the CONTENT-TYPE information object class. The set contains six information objects and is extensible, as indicated by the ellipsis (see Section 2.4, "Versioning and Extensibility").

ContentInfo is defined using type information from an information object, i.e., the type of the contentType field is that of the &id field from CONTENT-TYPE. An equivalent definition is as follows:

```
ContentType ::= OBJECT IDENTIFIER
```

Both fields in ContentInfo are constrained. The contentType field is constrained using a simple table constraint that restricts the values to those from the corresponding field of the objects in ContentSet. The content field is constrained using a component relationship constraint. Constraints are discussed in the next section.

2.2. Constraints

The X.68x versions of the ASN.1 specifications introduced the ability to use the object information sets as part of the constraint on the values that a field can take. Simple table constraints are used to restrict the set of values that can occur in a field. Component relation constraints allow for the restriction of a field based on contents of other fields in the type.

2.2.1. Simple Table Constraints

Simple table constraints are widely used in [RFC5911] and [RFC5912] to limit implementer options (although the constraints are almost always followed by or include extensibility markers, which make the parameters serve as information not as limitations). Table constraints are defined in [CCITT.X682.2002].

Some ASN.1 compilers have the ability to use the simple table constraint to check that a field contains one of the legal values.

The following example from [RFC5911] demonstrates using table constraints to clarify the intended usage of a particular field. The parameters indicate the types of attributes that are typically found in the signedAttrs and unsignedAttrs fields. In this example, the object sets are disjoint but this is not required. For example, in [RFC5912], there is some overlap between the CertExtensions and CrlExtensions sets.

```
-- from updated RFC 5652 module in [RFC5911]
SignerInfo ::= SEQUENCE {
    version CMSVersion,
    sid SignerIdentifier,
    digestAlgorithm DigestAlgorithmIdentifier,
    signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
    signatureAlgorithm SignatureAlgorithmIdentifier,
    signature SignatureValue,
    unsignedAttrs [1] IMPLICIT Attributes
         {{UnsignedAttributes}} OPTIONAL }

SignedAttributes ::= Attributes {{ SignedAttributesSet }}
```

```
SignedAttributesSet ATTRIBUTE ::=
       { aa-signingTime | aa-messageDigest | aa-contentType, ... }

UnsignedAttributes ATTRIBUTE ::= { aa-countersignature, ... }
```

2.2.2. Component Relation Constraints

Component relation constraints are often used to bind the type field of an open type to the identifier field. Using the binding in this way allows a compiler to immediately decode the associated type when the containing structure is defined. The following example from [RFC2560] as updated in [RFC5912] demonstrates this usage.

```
-- from updated RFC 2560 module in [RFC5912]
RESPONSE ::= TYPE-IDENTIFIER

ResponseSet RESPONSE ::= {basicResponse, ...}

ResponseBytes ::=       SEQUENCE {
    responseType        RESPONSE.
                            &id ({ResponseSet}),
    response            OCTET STRING (CONTAINING RESPONSE.
                            &Type({ResponseSet}{@responseType}))}
```

In this example, the response field is constrained to contain a type identified by the responseType field. The controlling field is identified using atNotation, e.g., "@responseType". atNotation can be defined relative to the outermost SEQUENCE, SET, or CHOICE or relative to the field with which the atNotation is associated. When there is no '.' immediately after the '@', the field appears as a member of the outermost SEQUENCE, SET, or CHOICE. When there is a '.' immediately after the '@', each '.' represents a SEQUENCE, SET, or CHOICE starting with the SEQUENCE, SET, or CHOICE that contains the field with which the atNotation is associated. For example, ResponseBytes could have been written as shown below. In this case, the syntax is very similar since the innermost and outermost SEQUENCE, SET, or CHOICE are the same.

```
ResponseBytes ::=       SEQUENCE {
    responseType        RESPONSE.
                            &id ({ResponseSet}),
    response            OCTET STRING (CONTAINING RESPONSE.
                            &Type({ResponseSet}{@.responseType}))}
```

The TaggedRequest example from [RFC5912] provides an example where the outermost and innermost SEQUENCE, SET, or CHOICE are different. Relative to the atNotation included in the definition of the

requestMessageValue field, the outermost SEQUENCE, SET, or CHOICE is TaggedRequest, and the innermost is the SEQUENCE used to define the orm field.

```
TaggedRequest ::= CHOICE {
   tcr               [0] TaggedCertificationRequest,
   crm               [1] CertReqMsg,
   orm               [2] SEQUENCE {
       bodyPartID            BodyPartID,
       requestMessageType    OTHER-REQUEST.&id({OtherRequests}),
       requestMessageValue   OTHER-REQUEST.&Type({OtherRequests}
                                 {@.requestMessageType})
   }
}
```

When referencing a field using atNotation, the definition of the field must be included within the outermost SEQUENCE, SET, or CHOICE. References to fields within structures that are defined separately are not allowed. For example, the following example includes invalid atNotation in the definition of the signature field within the SIGNED parameterized type.

```
AlgorithmIdentifier{ALGORITHM-TYPE, ALGORITHM-TYPE:AlgorithmSet} ::=
          SEQUENCE {
              algorithm   ALGORITHM-TYPE.&id({AlgorithmSet}),
              parameters  ALGORITHM-TYPE.
                     &Params({AlgorithmSet}{@algorithm}) OPTIONAL
          }

-- example containing invalid atNotation
SIGNED{ToBeSigned} ::= SEQUENCE {
   toBeSigned           ToBeSigned,
   algorithmIdentifier  AlgorithmIdentifier
                            { SIGNATURE-ALGORITHM, {...}}
   },
   signature BIT STRING (CONTAINING SIGNATURE-ALGORITHM.&Value(
                            {SignatureAlgorithms}
                            {@algorithmIdentifier.algorithm}))
}
```

Alternatively, the above example could be written with correct atNotation as follows, with the definition of the algorithm field included within ToBeSigned.

```
SIGNED{ToBeSigned} ::= SEQUENCE {
   toBeSigned           ToBeSigned,
   algorithmIdentifier  SEQUENCE {
       algorithm        SIGNATURE-ALGORITHM.
                            &id({SignatureAlgorithms}),
       parameters       SIGNATURE-ALGORITHM.
                            &Params({SignatureAlgorithms}
                                {@algorithmIdentifier.algorithm})
   },
   signature BIT STRING (CONTAINING SIGNATURE-ALGORITHM.&Value(
                            {SignatureAlgorithms}
                            {@algorithmIdentifier.algorithm}))
}
```

In the above example, the outermost SEQUENCE, SET, or CHOICE relative to the parameters field is the SIGNED parameterized type. The innermost structure is the SEQUENCE used as the type for the algorithmIdentifier field. The atNotation for the parameters field could be expressed using any of the following representations:

```
@algorithmIdentifier.algorithm

@.algorithm
```

The atNotation for the signature field has only one representation.

2.2.3. Content Constraints

Open types implemented as OCTET STRINGs or BIT STRINGs can be constrained using the contents constraints syntax defined in [CCITT.X682.2002]. Below are the revised definitions from [RFC5911] and [RFC5912]. These show usage of OCTET STRING and BIT STRING along with constrained sets of identifiers. The Extension definition uses a content constraint that requires the value of the OCTET STRING to be an encoding of the type associated with the information object selected from the ExtensionSet object set using the value of the extnID field. For reasons described in Section 2.2.2, "Component Relation Constraints", the SubjectPublicKeyInfo definition relies on prose to bind the BIT STRING to the type identifier because it is not possible to express a content constraint that includes a component relationship constraint to bind the type value within the algorithm field to the subjectPublicKey field.

```
-- from updated RFC 5280 module in [RFC5912]
Extension{EXTENSION:ExtensionSet} ::= SEQUENCE {
    extnID      EXTENSION.&id({ExtensionSet}),
    critical    BOOLEAN
    -- (EXTENSION.&Critical({ExtensionSet}{@extnID}))
                        DEFAULT FALSE,
    extnValue   OCTET STRING (CONTAINING
                  EXTENSION.&ExtnType({ExtensionSet}{@extnID}))
                  --  contains the DER encoding of the ASN.1 value
                  --  corresponding to the extension type identified
                  --  by extnID
}

SubjectPublicKeyInfo  ::=  SEQUENCE  {
    algorithm            AlgorithmIdentifier{PUBLIC-KEY,
                             {PublicKeyAlgorithms}},
    subjectPublicKey     BIT STRING
}
```

2.3. Parameterization

Parameterization is defined in [CCITT.X683.2002] and can also be used to define new types in a way similar to the macro notation described in Annex A of X.208. The following example from [RFC5912] shows this usage. The toBeSigned field takes the type passed as a parameter.

```
-- from [RFC5912]
SIGNED{ToBeSigned} ::= SEQUENCE {
    toBeSigned  ToBeSigned,
    algorithm   AlgorithmIdentifier{SIGNATURE-ALGORITHM,
                    {SignatureAlgorithms}},
    signature   BIT STRING
}

-- from updated RFC5280 module in [RFC5912]
Certificate  ::=  SIGNED{TBSCertificate}
```

Parameters need not be simple types. The following example demonstrates the usage of an information object class and an information object set as parameters. The first parameter in the definition of AlgorithmIdentifier is an information object class. Information object classes used for this parameter must have &id and &Params fields, which determine the type of the algorithm and parameters fields. Other fields may be present in the information object class, but they are not used by the definition of AlgorithmIdentifier, as demonstrated by the SIGNATURE-ALGORITHM class

shown below. The second parameter is an information object set that is used to constrain the values that appear in the algorithm and parameters fields.

```
-- from [RFC5912]
AlgorithmIdentifier{ALGORITHM-TYPE, ALGORITHM-TYPE:AlgorithmSet}
    ::= SEQUENCE
{
    algorithm   ALGORITHM-TYPE.&id({AlgorithmSet}),
    parameters  ALGORITHM-TYPE.&Params
                  ({AlgorithmSet}{@algorithm}) OPTIONAL
}

SIGNATURE-ALGORITHM ::= CLASS {
    &id             OBJECT IDENTIFIER,
    &Params         OPTIONAL,
    &Value          OPTIONAL,
    &paramPresence  ParamOptions DEFAULT absent,
    &HashSet        DIGEST-ALGORITHM OPTIONAL,
    &PublicKeySet   PUBLIC-KEY OPTIONAL,
    &smimeCaps      SMIME-CAPS OPTIONAL
} WITH SYNTAX {
    IDENTIFIER &id
    [VALUE &Value]
    [PARAMS [TYPE &Params] ARE &paramPresence ]
    [HASHES &HashSet]
    [PUBLIC KEYS &PublicKeySet]
    [SMIME CAPS &smimeCaps]
}

-- from updated RFC 2560 module in [RFC5912]
BasicOCSPResponse       ::= SEQUENCE {
    tbsResponseData      ResponseData,
    signatureAlgorithm   AlgorithmIdentifier{SIGNATURE-ALGORITHM,
                          {sa-dsaWithSHA1 | sa-rsaWithSHA1 |
                               sa-rsaWithMD5 | sa-rsaWithMD2, ...}},
    signature            BIT STRING,
    certs            [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL
}
```

2.4. Versioning and Extensibility

Specifications are often revised and ASN.1 modules updated to include new components. [CCITT.X681.2002] provides two mechanisms useful in supporting extensibility: extension markers and version brackets.

2.4.1. Extension Markers

An extension marker is represented by an ellipsis (i.e., three adjacent periods). Extension markers are included in specifications at points where the protocol designer anticipates future changes. This can also be achieved by including EXTENSIBILITY IMPLIED in the ASN.1 module definition. EXTENSIBILITY IMPLIED is the equivalent to including an extension marker in each type defined in the ASN.1 module. Extensibility markers are used throughout [RFC5911] and [RFC5912] where object sets are defined. In other instances, the updated modules retroactively added extension markers where fields were added to an earlier version by an update, as shown in the CertificateChoices example below.

Examples:

```
-- from updated RFC 3370 module in [RFC5911]
KeyAgreementAlgs KEY-AGREE ::= { kaa-esdh | kaa-ssdh, ...}

-- from updated RFC 5652 module in [RFC5911]
CertificateChoices ::= CHOICE {
    certificate Certificate,
    extendedCertificate [0] IMPLICIT ExtendedCertificate,
         -- Obsolete
    ...,
    [[3: v1AttrCert [1] IMPLICIT AttributeCertificateV1]],
         -- Obsolete
    [[4: v2AttrCert [2] IMPLICIT AttributeCertificateV2]],
    [[5: other      [3] IMPLICIT OtherCertificateFormat]]
}
```

Protocol designers should use extension markers within definitions that are likely to change. For example, extensibility markers should be used when enumerating error values.

2.4.2. Version Brackets

Version brackets can be used to indicate features that are available in later versions of an ASN.1 module but not in earlier versions. [RFC5912] added version brackets to the definition of TBSCertificate to illustrate the addition of the issuerUniqueID, subjectUniqueID, and extensions fields, as shown below.

```
-- from updated RFC 5280 module in [RFC5912]
TBSCertificate  ::=  SEQUENCE  {
    version         [0]  Version DEFAULT v1,
    serialNumber         CertificateSerialNumber,
    signature            AlgorithmIdentifier{SIGNATURE-ALGORITHM,
                              {SignatureAlgorithms}},
    issuer               Name,
    validity             Validity,
    subject              Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    ... ,
    [[2:                -- If present, version MUST be v2
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL
    ]],
    [[3:                -- If present, version MUST be v3 --
    extensions      [3]  ExtensionSet{{CertExtensions}} OPTIONAL
    ]], ... }
```

3. Character Set Differences

X.68s uses a character set that is a superset of the character set defined in X.208. The character set defined in X.208 includes the following:

A to Z

a to z

0 to 9

:=,{}<.

()[]-'"

The character set in X.68x additionally includes the following:

!&*/;>@^_|

The > and | characters can also be used in X.208 syntax in macro definitions.

4. ASN.1 Translation

4.1. Downgrading from X.68x to X.208

At a minimum, downgrading an ASN.1 module from X.68x syntax to X.208 requires the removal of features not supported by X.208. As indicated above, the features most commonly used in IETF Security Area ASN.1 modules are information object classes (and object sets), content constraints, parameterization, extension markers, and version brackets. Extension markers and version brackets can simply be deleted (or commented out). The definitions for information object classes and object sets can also be deleted or commented out, as these will not be used. The following checklist can be used in most cases:

- Remove all Information Set Class, Information Set Object, and Information Set Object Set definitions and imports from the file.
- Replace all fixed Type Information Set Class element references with the fixed type. (That is, replace FOO.&id with OBJECT IDENTIFIER.)
- Delete all simple constraints.
- Delete all CONTAINING statements.
- Replace all variable Type Information Set Class element references with either ANY or ANY DEFINED BY statements.
- Remove version and extension markers.
- Manually enforce all instances of parameterized types.

4.2. Upgrading from X.208 to X.68x

The amount of change associated with upgrading from X.208 syntax to X.68x syntax is dependent on the reasons for changing and personal style. A minimalist approach could consist of altering any deprecated features, most commonly ANY DEFINED BY, and adding any necessary extensibility markers. A more comprehensive approach may include the introduction of constraints, parameterization, versioning, extensibility, etc.

The following checklist can be used when upgrading a module without introducing constraints:

Use TYPE-IDENTIFIER.&Type for "ANY".

Use TYPE-IDENTIFIER.&Type for "ANY DEFINED BY ...".

When constraints are introduced during an upgrade, additional steps are necessary:

1. Identify each unique class that should be defined based on what types of things exist.

2. Define an Information Object Class for each of the classes above with the appropriate elements.

3. Define all of the appropriate Information Object Sets based on the classes defined in step 2 along with the different places that they should be used.

4. Replace ANY by the appropriate class and variable type element.

5. Replace ANY DEFINED BY with the appropriate variable type element and the components constraint. Replace the element used in the constraint with the appropriate fixed type element and simple constraint.

6. Add any simple constraints as appropriate.

7. Define any objects and fill in elements for object sets as appropriate.

5. Security Considerations

Where a module is downgraded from X.68x syntax to X.208 there is loss of potential automated enforcement of constraints expressed by the author of the module being downgraded. These constraints should be captured in prose or ASN.1 comments and enforced through other means, as necessary.

Depending on the feature set of the ASN.1 compiler being used, the code to enforce and use constraints may be generated automatically or may require the programmer to do this independently. It is the responsibility of the programmer to ensure that the constraints on the ASN.1 expressed either in prose or in the ASN.1 module are actually enforced.

6. References

6.1. Normative References

[CCITT.X208.1988] International Telephone and Telegraph Consultative Committee, "Specification of Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.208, November 1988.

[CCITT.X680.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.

[CCITT.X681.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Information object specification", CCITT Recommendation X.681, July 2002.

[CCITT.X682.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Constraint specification", CCITT Recommendation X.682, July 2002.

[CCITT.X683.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications", CCITT Recommendation X.683, July 2002.

6.2. Informative References

[CCITT.X209.1988] International Telephone and Telegraph Consultative Committee, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.209, 1988.

[CCITT.X690.2002] International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

[RFC2560] Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 2560, June 1999.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[RFC5911] Hoffman, P. and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, June 2010.

[RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, June 2010.

Authors' Addresses

Carl Wallace
Cygnacom Solutions
Suite 5400
7925 Jones Branch Drive
McLean, VA 22102

EMail: cwallace@cygnacom.com

Charles Gardiner
BBN Technologies
10 Moulton Street
Cambridge, MA 02138

EMail: gardiner@bbn.com