Network Working Group   S. Legg
Request for Comments: 4911   eB2Bcom
Category: Experimental   July 2007

                   Encoding Instructions for the
                 Robust XML Encoding Rules (RXER)

Abstract

   This document defines encoding instructions that may be used in an
   Abstract Syntax Notation One (ASN.1) specification to alter how ASN.1
   values are encoded by the Robust XML Encoding Rules (RXER) and
   Canonical Robust XML Encoding Rules (CRXER), for example, to encode a
   component of an ASN.1 value as an Extensible Markup Language (XML)
   attribute rather than as a child element. Some of these encoding
   instructions also affect how an ASN.1 specification is translated
   into an Abstract Syntax Notation X (ASN.X) specification. Encoding
   instructions that allow an ASN.1 specification to reference
   definitions in other XML schema languages are also defined.

Table of Contents

1. Introduction

   This document defines encoding instructions [X.680-1] that may be
   used in an Abstract Syntax Notation One (ASN.1) [X.680] specification
   to alter how ASN.1 values are encoded by the Robust XML Encoding
   Rules (RXER) [RXER] and Canonical Robust XML Encoding Rules (CRXER)
   [RXER], for example, to encode a component of an ASN.1 value as an
   Extensible Markup Language (XML) [XML10] attribute rather than as a
   child element. Some of these encoding instructions also affect how
   an ASN.1 specification is translated into an Abstract Syntax Notation
   X (ASN.X) specification [ASN.X].

   This document also defines encoding instructions that allow an ASN.1
   specification to incorporate the definitions of types, elements, and
   attributes in specifications written in other XML schema languages.
   References to XML Schema [XSD1] types, elements, and attributes,
   RELAX NG [RNG] named patterns and elements, and XML document type
   definition (DTD) [XML10] element types are supported.

   In most cases, the effect of an encoding instruction is only briefly
   mentioned in this document. The precise effects of these encoding
   instructions are described fully in the specifications for RXER
   [RXER] and ASN.X [ASN.X], at the points where they apply.

2. Conventions

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED" and "MAY" in this document are
   to be interpreted as described in BCP 14, RFC 2119 [BCP14]. The key
   word "OPTIONAL" is exclusively used with its ASN.1 meaning.

   Throughout this document "type" shall be taken to mean an ASN.1 type,
   and "value" shall be taken to mean an ASN.1 abstract value, unless
   qualified otherwise.

   A reference to an ASN.1 production [X.680] (e.g., Type, NamedType) is
   a reference to text in an ASN.1 specification corresponding to that
   production. Throughout this document, "component" is synonymous with
   NamedType.

   This document uses the namespace prefix "xsi:" to stand for the
   namespace name [XMLNS10] "http://www.w3.org/2001/XMLSchema-instance".

   Example ASN.1 definitions in this document are assumed to be defined
   in an ASN.1 module with a TagDefault of "AUTOMATIC TAGS" and an
   EncodingReferenceDefault [X.680-1] of "RXER INSTRUCTIONS".

3. Definitions

   The following definition of base type is used in specifying a number
   of encoding instructions.

   Definition (base type): If a type, T, is a constrained type, then the
   base type of T is the base type of the type that is constrained; else
   if T is a prefixed type, then the base type of T is the base type of
   the type that is prefixed; else if T is a type notation that
   references or denotes another type (i.e., DefinedType,
   ObjectClassFieldType, SelectionType, TypeFromObject, or
   ValueSetFromObjects), then the base type of T is the base type of the
   type that is referenced or denoted; otherwise, the base type of T is
   T itself.

      Aside: A tagged type is a special case of a prefixed type.

4. Notation for RXER Encoding Instructions

   The grammar of ASN.1 permits the application of encoding instructions
   [X.680-1], through type prefixes and encoding control sections, that
   modify how abstract values are encoded by nominated encoding rules.

   The generic notation for type prefixes and encoding control sections
   is defined by the ASN.1 basic notation [X.680] [X.680-1], and
   includes an encoding reference to identify the specific encoding
   rules that are affected by the encoding instruction.

   The encoding reference that identifies the Robust XML Encoding rules
   is literally RXER. An RXER encoding instruction applies equally to
   both RXER and CRXER encodings.

   The specific notation for an encoding instruction for a specific set
   of encoding rules is left to the specification of those encoding
   rules. Consequently, this companion document to the RXER
   specification [RXER] defines the notation for RXER encoding
   instructions. Specifically, it elaborates the EncodingInstruction
   and EncodingInstructionAssignmentList placeholder productions of the
   ASN.1 basic notation.

   In the context of the RXER encoding reference, the
   EncodingInstruction production is defined as follows, using the
   conventions of the ASN.1 basic notation:

```
   EncodingInstruction ::=
       AttributeInstruction |
       AttributeRefInstruction |
       ComponentRefInstruction |
       ElementRefInstruction |
       GroupInstruction |
       InsertionsInstruction |
       ListInstruction |
       NameInstruction |
       RefAsElementInstruction |
       RefAsTypeInstruction |
       SimpleContentInstruction |
       TypeAsVersionInstruction |
       TypeRefInstruction |
       UnionInstruction |
       ValuesInstruction |
       VersionIndicatorInstruction
```

   In the context of the RXER encoding reference, the
   EncodingInstructionAssignmentList production (which only appears in
   an encoding control section) is defined as follows:

```
   EncodingInstructionAssignmentList ::=
       SchemaIdentityInstruction ?
       TargetNamespaceInstruction ?
       TopLevelComponents ?

   TopLevelComponents ::= TopLevelComponent TopLevelComponents ?

   TopLevelComponent ::= "COMPONENT" NamedType
```

   Definition (top-level NamedType): A NamedType is a top-level
   NamedType (equivalently, a top-level component) if and only if it is
   the NamedType in a TopLevelComponent. A NamedType nested within the
   Type of the NamedType of a TopLevelComponent is not itself a
   top-level NamedType.

      Aside: Specification writers should note that non-trivial types
      defined within a top-level NamedType will not be visible to ASN.1
      tools that do not understand RXER.

   Although a top-level NamedType only appears in an RXER encoding
   control section, the default encoding reference for the module
   [X.680-1] still applies when parsing a top-level NamedType.

   Each top-level NamedType within a module SHALL have a distinct
   identifier.

The NamedType production is defined by the ASN.1 basic notation. The
other productions are described in subsequent sections and make use
of the following productions:

    NCNameValue ::= Value

    AnyURIValue ::= Value

    QNameValue ::= Value

    NameValue ::= Value

The Value production is defined by the ASN.1 basic notation.

The governing type for the Value in an NCNameValue is the NCName type
from the AdditionalBasicDefinitions module [RXER].

The governing type for the Value in an AnyURIValue is the AnyURI type
from the AdditionalBasicDefinitions module.

The governing type for the Value in a QNameValue is the QName type
from the AdditionalBasicDefinitions module.

The governing type for the Value in a NameValue is the Name type from
the AdditionalBasicDefinitions module.

The Value in an NCNameValue, AnyURIValue, QNameValue, or NameValue
SHALL NOT be a DummyReference [X.683] and SHALL NOT textually contain
a nested DummyReference.

    Aside: Thus, encoding instructions are not permitted to be
    parameterized in any way. This restriction will become important
    if a future specification for ASN.X explicitly represents
    parameterized definitions and parameterized references instead of
    expanding out parameterized references as in the current
    specification. A parameterized definition could not be directly
    translated into ASN.X if it contained encoding instructions that
    were not fully specified.

5. Component Encoding Instructions

    Certain of the RXER encoding instructions are categorized as
    component encoding instructions. The component encoding instructions
    are the ATTRIBUTE, ATTRIBUTE-REF, COMPONENT-REF, GROUP, ELEMENT-REF,
    NAME, REF-AS-ELEMENT, SIMPLE-CONTENT, TYPE-AS-VERSION, and
    VERSION-INDICATOR encoding instructions (whose notations are
    described respectively by AttributeInstruction,
    AttributeRefInstruction, ComponentRefInstruction, GroupInstruction,

ElementRefInstruction, NameInstruction, RefAsElementInstruction,
SimpleContentInstruction, TypeAsVersionInstruction, and
VersionIndicatorInstruction).

The Type in the EncodingPrefixedType for a component encoding
instruction SHALL be either:

(1) the Type in a NamedType, or

(2) the Type in an EncodingPrefixedType in a PrefixedType in a
    BuiltinType in a Type that is one of (1) to (4), or

(3) the Type in an TaggedType in a PrefixedType in a BuiltinType in a
    Type that is one of (1) to (4), or

(4) the Type in a ConstrainedType (excluding a TypeWithConstraint) in
    a Type that is one of (1) to (4).

   Aside: The effect of this condition is to force the component
   encoding instructions to be textually within the NamedType to
   which they apply. Only case (2) can be true on the first
   iteration as the Type belongs to an EncodingPrefixedType; however,
   any of (1) to (4) can be true on subsequent iterations.

Case (4) is not permitted when the encoding instruction is the
ATTRIBUTE-REF, COMPONENT-REF, ELEMENT-REF, or REF-AS-ELEMENT encoding
instruction.

The NamedType in case (1) is said to be "subject to" the component
encoding instruction.

A top-level NamedType SHALL NOT be subject to an ATTRIBUTE-REF,
COMPONENT-REF, GROUP, ELEMENT-REF, REF-AS-ELEMENT, or SIMPLE-CONTENT
encoding instruction.

   Aside: This condition does not preclude these encoding
   instructions being used on a nested NamedType.

A NamedType SHALL NOT be subject to two or more component encoding
instructions of the same kind, e.g., a NamedType is not permitted to
be subject to two NAME encoding instructions.

The ATTRIBUTE, ATTRIBUTE-REF, COMPONENT-REF, GROUP, ELEMENT-REF,
REF-AS-ELEMENT, SIMPLE-CONTENT, and TYPE-AS-VERSION encoding
instructions are mutually exclusive. The NAME, ATTRIBUTE-REF,
COMPONENT-REF, ELEMENT-REF, and REF-AS-ELEMENT encoding instructions
are mutually exclusive. A NamedType SHALL NOT be subject to two or
more encoding instructions that are mutually exclusive.

A SelectionType [X.680] SHALL NOT be used to select the Type from a
NamedType that is subject to an ATTRIBUTE-REF, COMPONENT-REF,
ELEMENT-REF or REF-AS-ELEMENT encoding instruction. The other
component encoding instructions are not inherited by the type denoted
by a SelectionType.

Definition (attribute component): An attribute component is a
NamedType that is subject to an ATTRIBUTE or ATTRIBUTE-REF encoding
instruction, or subject to a COMPONENT-REF encoding instruction that
references a top-level NamedType that is subject to an ATTRIBUTE
encoding instruction.

Definition (element component): An element component is a NamedType
that is not subject to an ATTRIBUTE, ATTRIBUTE-REF, GROUP, or
SIMPLE-CONTENT encoding instruction, and not subject to a
COMPONENT-REF encoding instruction that references a top-level
NamedType that is subject to an ATTRIBUTE encoding instruction.

   Aside: A NamedType subject to a GROUP or SIMPLE-CONTENT encoding
   instruction is neither an attribute component nor an element
   component.

6. Reference Encoding Instructions

Certain of the RXER encoding instructions are categorized as
reference encoding instructions. The reference encoding instructions
are the ATTRIBUTE-REF, COMPONENT-REF, ELEMENT-REF, REF-AS-ELEMENT,
REF-AS-TYPE, and TYPE-REF encoding instructions (whose notations are
described respectively by AttributeRefInstruction,
ComponentRefInstruction, ElementRefInstruction,
RefAsElementInstruction, RefAsTypeInstruction, and
TypeRefInstruction). These encoding instructions (except
COMPONENT-REF) allow an ASN.1 specification to incorporate the
definitions of types, elements, and attributes in specifications
written in other XML schema languages, through implied constraints on
the markup that may appear in values of the Markup ASN.1 type from
the AdditionalBasicDefinitions module [RXER] (for ELEMENT-REF,
REF-AS-ELEMENT, REF-AS-TYPE, and TYPE-REF) or the UTF8String type
(for ATTRIBUTE-REF). References to XML Schema [XSD1] types,
elements, and attributes, RELAX NG [RNG] named patterns and elements,
and XML document type definition (DTD) [XML10] element types are
supported. References to ASN.1 types and top-level components are
also permitted. The COMPONENT-REF encoding instruction provides a
more direct method of referencing a top-level component.

The Type in the EncodingPrefixedType for an ELEMENT-REF,
REF-AS-ELEMENT, REF-AS-TYPE, or TYPE-REF encoding instruction SHALL
be either:

   (1) a ReferencedType that is a DefinedType that is a typereference
       (not a DummyReference) or ExternalTypeReference that references
       the Markup ASN.1 type from the AdditionalBasicDefinitions module
       [RXER], or

   (2) a BuiltinType that is a PrefixedType that is a TaggedType where
       the Type in the TaggedType is one of (1) to (3), or

   (3) a BuiltinType that is a PrefixedType that is an
       EncodingPrefixedType where the Type in the EncodingPrefixedType
       is one of (1) to (3) and the EncodingPrefix in the
       EncodingPrefixedType does not contain a reference encoding
       instruction.

      Aside: Case (3) and similar cases for the ATTRIBUTE-REF and
      COMPONENT-REF encoding instructions have the effect of making the
      reference encoding instructions mutually exclusive as well as
      singly occurring.

   With respect to the REF-AS-TYPE and TYPE-REF encoding instructions,
   the DefinedType in case (1) is said to be "subject to" the encoding
   instruction.

   The restrictions on the Type in the EncodingPrefixedType for an
   ATTRIBUTE-REF encoding instruction are specified in Section 9. The
   restrictions on the Type in the EncodingPrefixedType for a
   COMPONENT-REF encoding instruction are specified in Section 10.

   The reference encoding instructions make use of a common production
   defined as follows:

      RefParameters ::= ContextParameter ?

      ContextParameter ::= "CONTEXT" AnyURIValue

   A RefParameters instance provides extra information about a reference
   to a definition. A ContextParameter is used when a reference is
   ambiguous, i.e., refers to definitions in more than one schema
   document or external DTD subset. This situation would occur, for
   example, when importing types with the same name from independently
   developed XML Schemas defined without a target namespace [XSD1].
   When used in conjunction with a reference to an element type in an
   external DTD subset, the AnyURIValue in the ContextParameter is the
   system identifier (a Uniform Resource Identifier or URI [URI]) of the
   external DTD subset; otherwise, the AnyURIValue is a URI that
   indicates the intended schema document, either an XML Schema
   specification, a RELAX NG specification, or an ASN.1 or ASN.X
   specification.

7. Expanded Names of Components

   Each NamedType has an associated expanded name [XMLNS10], determined
   as follows:

   (1) if the NamedType is subject to a NAME encoding instruction, then
       the local name of the expanded name is the character string
       specified by the NCNameValue of the NAME encoding instruction,

   (2) else if the NamedType is subject to a COMPONENT-REF encoding
       instruction, then the expanded name is the same as the expanded
       name of the referenced top-level NamedType,

   (3) else if the NamedType is subject to an ATTRIBUTE-REF or
       ELEMENT-REF encoding instruction, then the namespace name of the
       expanded name is equal to the namespace-name component of the
       QNameValue of the encoding instruction, and the local name is
       equal to the local-name component of the QNameValue,

   (4) else if the NamedType is subject to a REF-AS-ELEMENT encoding
       instruction, then the local name of the expanded name is the
       LocalPart [XMLNS10] of the qualified name specified by the
       NameValue of the encoding instruction,

   (5) otherwise, the local name of the expanded name is the identifier
       of the NamedType.

   In cases (1) and (5), if the NamedType is a top-level NamedType and
   the module containing the NamedType has a TARGET-NAMESPACE encoding
   instruction, then the namespace name of the expanded name is the
   character string specified by the AnyURIValue of the TARGET-NAMESPACE
   encoding instruction; otherwise, the namespace name has no value.

      Aside: Thus, the TARGET-NAMESPACE encoding instruction applies to
      a top-level NamedType but not to any other NamedType.

   In case (4), if the encoding instruction contains a Namespace, then
   the namespace name of the expanded name is the character string
   specified by the AnyURIValue of the Namespace; otherwise, the
   namespace name has no value.

   The expanded names for the attribute components of a CHOICE,
   SEQUENCE, or SET type MUST be distinct. The expanded names for the
   components of a CHOICE, SEQUENCE, or SET type that are not attribute
   components MUST be distinct. These tests are applied after the
   COMPONENTS OF transformation specified in X.680, Clause 24.4 [X.680].

Aside: Two components of the same CHOICE, SEQUENCE, or SET type
may have the same expanded name if one of them is an attribute
component and the other is not. Note that the "not" case includes
components that are subject to a GROUP or SIMPLE-CONTENT encoding
instruction.

The expanded name of a top-level NamedType subject to an ATTRIBUTE
encoding instruction MUST be distinct from the expanded name of every
other top-level NamedType subject to an ATTRIBUTE encoding
instruction in the same module.

The expanded name of a top-level NamedType not subject to an
ATTRIBUTE encoding instruction MUST be distinct from the expanded
name of every other top-level NamedType not subject to an ATTRIBUTE
encoding instruction in the same module.

Aside: Two top-level components may have the same expanded name if
one of them is an attribute component and the other is not.

8. The ATTRIBUTE Encoding Instruction

The ATTRIBUTE encoding instruction causes an RXER encoder to encode a
value of the component to which it is applied as an XML attribute
instead of as a child element.

The notation for an ATTRIBUTE encoding instruction is defined as
follows:

    AttributeInstruction ::= "ATTRIBUTE"

The base type of the type of a NamedType that is subject to an
ATTRIBUTE encoding instruction SHALL NOT be:

(1) a CHOICE, SET, or SET OF type, or

(2) a SEQUENCE type other than the one defining the QName type from
    the AdditionalBasicDefinitions module [RXER] (i.e., QName is
    allowed), or

(3) a SEQUENCE OF type where the SequenceOfType is not subject to a
    LIST encoding instruction, or

(4) an open type.

    Example

        PersonalDetails ::= SEQUENCE {
            firstName   [ATTRIBUTE] UTF8String,
            middleName   [ATTRIBUTE] UTF8String,
            surname   [ATTRIBUTE] UTF8String
        }

9. The ATTRIBUTE-REF Encoding Instruction

    The ATTRIBUTE-REF encoding instruction causes an RXER encoder to
    encode a value of the component to which it is applied as an XML
    attribute instead of as a child element, where the attribute's name
    is a qualified name of the attribute declaration referenced by the
    encoding instruction. In addition, the ATTRIBUTE-REF encoding
    instruction causes values of the UTF8String type to be restricted to
    conform to the type of the attribute declaration.

    The notation for an ATTRIBUTE-REF encoding instruction is defined as
    follows:

        AttributeRefInstruction ::=
            "ATTRIBUTE-REF" QNameValue RefParameters

    Taken together, the QNameValue and the ContextParameter in the
    RefParameters (if present) MUST reference an XML Schema attribute
    declaration or a top-level NamedType that is subject to an ATTRIBUTE
    encoding instruction.

    The type of a referenced XML Schema attribute declaration SHALL NOT
    be, either directly or by derivation, the XML Schema type QName,
    NOTATION, ENTITY, ENTITIES, or anySimpleType.

        Aside: Values of these types require information from the context
        of the attribute for interpretation. Because an ATTRIBUTE-REF
        encoding instruction is restricted to prefixing the ASN.1
        UTF8String type, there is no mechanism to capture such context.

    The type of a referenced top-level NamedType SHALL NOT be, either
    directly or by subtyping, the QName type from the
    AdditionalBasicDefinitions module [RXER].

    The Type in the EncodingPrefixedType for an ATTRIBUTE-REF encoding
    instruction SHALL be either:

    (1) the UTF8String type, or

   (2) a BuiltinType that is a PrefixedType that is a TaggedType where
       the Type in the TaggedType is one of (1) to (3), or

   (3) a BuiltinType that is a PrefixedType that is an
       EncodingPrefixedType where the Type in the EncodingPrefixedType
       is one of (1) to (3) and the EncodingPrefix in the
       EncodingPrefixedType does not contain a reference encoding
       instruction.

   The identifier of a NamedType subject to an ATTRIBUTE-REF encoding
   instruction does not contribute to the name of attributes in an RXER
   encoding. For the sake of consistency, the identifier SHOULD, where
   possible, be the same as the local name of the referenced attribute
   declaration.

10. The COMPONENT-REF Encoding Instruction

   The ASN.1 basic notation does not have a concept of a top-level
   NamedType and therefore does not have a mechanism to reference a
   top-level NamedType. The COMPONENT-REF encoding instruction provides
   a way to specify that a NamedType within a combining type definition
   is equivalent to a referenced top-level NamedType.

   The notation for a COMPONENT-REF encoding instruction is defined as
   follows:

       ComponentRefInstruction ::= "COMPONENT-REF" ComponentReference

       ComponentReference ::=
           InternalComponentReference |
           ExternalComponentReference

       InternalComponentReference ::= identifier FromModule ?

       FromModule ::= "FROM" GlobalModuleReference

       ExternalComponentReference ::= modulereference "." identifier

   The GlobalModuleReference production is defined by the ASN.1 basic
   notation [X.680]. If the GlobalModuleReference is absent from an
   InternalComponentReference, then the identifier MUST be the
   identifier of a top-level NamedType in the same module. If the
   GlobalModuleReference is present in an InternalComponentReference,
   then the identifier MUST be the identifier of a top-level NamedType
   in the referenced module.

The modulereference in an ExternalComponentReference is used in the
same way as a modulereference in an ExternalTypeReference. The
identifier in an ExternalComponentReference MUST be the identifier of
a top-level NamedType in the referenced module.

The Type in the EncodingPrefixedType for a COMPONENT-REF encoding
instruction SHALL be either:

(1) a ReferencedType that is a DefinedType that is a typereference
    (not a DummyReference) or an ExternalTypeReference, or

(2) a BuiltinType or ReferencedType that is one of the productions in
    Table 1 in Section 5 of the specification for RXER [RXER], or

(3) a BuiltinType that is a PrefixedType that is a TaggedType where
    the Type in the TaggedType is one of (1) to (4), or

(4) a BuiltinType that is a PrefixedType that is an
    EncodingPrefixedType where the Type in the EncodingPrefixedType
    is one of (1) to (4) and the EncodingPrefix in the
    EncodingPrefixedType does not contain a reference encoding
    instruction.

The restrictions on the use of RXER encoding instructions are such
that no other RXER encoding instruction is permitted within a
NamedType if the NamedType is subject to a COMPONENT-REF encoding
instruction.

The Type in the top-level NamedType referenced by the COMPONENT-REF
encoding instruction MUST be either:

(a) if the preceding case (1) is used, a ReferencedType that is a
    DefinedType that is a typereference or ExternalTypeReference that
    references the same type as the DefinedType in case (1), or

(b) if the preceding case (2) is used, a BuiltinType or
    ReferencedType that is the same as the BuiltinType or
    ReferencedType in case (2), or

(c) a BuiltinType that is a PrefixedType that is an
    EncodingPrefixedType where the Type in the EncodingPrefixedType
    is one of (a) to (c), and the EncodingPrefix in the
    EncodingPrefixedType contains an RXER encoding instruction.

In principle, the COMPONENT-REF encoding instruction creates a
notional NamedType where the expanded name is that of the referenced
top-level NamedType and the Type in case (1) or (2) is substituted by
the Type of the referenced top-level NamedType.

In practice, it is sufficient for non-RXER encoders and decoders to
use the original NamedType rather than the notional NamedType because
the Type in case (1) or (2) can only differ from the Type of the
referenced top-level NamedType by having fewer RXER encoding
instructions, and RXER encoding instructions are ignored by non-RXER
encoders and decoders.

Although any prefixes for the Type in case (1) or (2) would be
bypassed, it is sufficient for RXER encoders and decoders to use the
referenced top-level NamedType instead of the notional NamedType
because these prefixes cannot be RXER encoding instructions (except,
of course, for the COMPONENT-REF encoding instruction) and can have
no effect on an RXER encoding.

Example

```
Modules ::= SEQUENCE OF
    module [COMPONENT-REF module
                FROM AbstractSyntaxNotation-X
                    { 1 3 6 1 4 1 21472 1 0 1 }]
                ModuleDefinition
```

Note that the "module" top-level NamedType in the
AbstractSyntaxNotation-X module is defined like so:

```
COMPONENT module ModuleDefinition
```

The ASN.X translation of the SEQUENCE OF type definition provides
a more natural representation:

```
<namedType xmlns:asnx="urn:ietf:params:xml:ns:asnx"
            name="Modules">
 <sequenceOf>
  <element ref="asnx:module"/>
 </sequenceOf>
</namedType>
```

Aside: The <namedType> element in ASN.X corresponds to a
TypeAssignment, not a NamedType.

The identifier of a NamedType subject to a COMPONENT-REF encoding
instruction does not contribute to an RXER encoding. For the sake of
consistency with other encoding rules, the identifier SHOULD be the
same as the identifier in the ComponentRefInstruction.

11. The ELEMENT-REF Encoding Instruction

   The ELEMENT-REF encoding instruction causes an RXER encoder to encode
   a value of the component to which it is applied as an element where
   the element's name is a qualified name of the element declaration
   referenced by the encoding instruction. In addition, the ELEMENT-REF
   encoding instruction causes values of the Markup ASN.1 type to be
   restricted to conform to the type of the element declaration.

   The notation for an ELEMENT-REF encoding instruction is defined as
   follows:

      ElementRefInstruction ::= "ELEMENT-REF" QNameValue RefParameters

   Taken together, the QNameValue and the ContextParameter in the
   RefParameters (if present) MUST reference an XML Schema element
   declaration, a RELAX NG element definition, or a top-level NamedType
   that is not subject to an ATTRIBUTE encoding instruction.

   A referenced XML Schema element declaration MUST NOT have a type that
   requires the presence of values for the XML Schema ENTITY or ENTITIES
   types.

      Aside: Entity declarations are not supported by CRXER.

   Example

      AnySchema ::= CHOICE {
          module   [ELEMENT-REF {
                      namespace-name
                          "urn:ietf:params:xml:ns:asnx",
                      local-name "module" }]
                   Markup,
          schema   [ELEMENT-REF {
                      namespace-name
                          "http://www.w3.org/2001/XMLSchema",
                      local-name "schema" }]
                   Markup,
          grammar [ELEMENT-REF {
                      namespace-name
                          "http://relaxng.org/ns/structure/1.0",
                      local-name "grammar" }]
                   Markup
      }

      The ASN.X translation of the choice type definition provides a
      more natural representation:

```
      <namedType xmlns:asnx="urn:ietf:params:xml:ns:asnx"
                 xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns:rng="http://relaxng.org/ns/structure/1.0"
                 name="AnySchema">
       <choice>
        <element ref="asnx:module" embedded="true"/>
        <element ref="xs:schema" embedded="true"/>
        <element ref="rng:grammar" embedded="true"/>
       </choice>
      </namedType>
```

   The identifier of a NamedType subject to an ELEMENT-REF encoding
   instruction does not contribute to the name of an element in an RXER
   encoding. For the sake of consistency, the identifier SHOULD, where
   possible, be the same as the local name of the referenced element
   declaration.

12. The LIST Encoding Instruction

   The LIST encoding instruction causes an RXER encoder to encode a
   value of a SEQUENCE OF type as a white-space-separated list of the
   component values.

   The notation for a LIST encoding instruction is defined as follows:

```
      ListInstruction ::= "LIST"
```

   The Type in an EncodingPrefixedType for a LIST encoding instruction
   SHALL be either:

   (1) a BuiltinType that is a SequenceOfType of the
       "SEQUENCE OF NamedType" form, or

   (2) a ConstrainedType that is a TypeWithConstraint of the
       "SEQUENCE Constraint OF NamedType" form or
       "SEQUENCE SizeConstraint OF NamedType" form, or

   (3) a ConstrainedType that is not a TypeWithConstraint where the Type
       in the ConstrainedType is one of (1) to (5), or

   (4) a BuiltinType that is a PrefixedType that is a TaggedType where
       the Type in the TaggedType is one of (1) to (5), or

   (5) a BuiltinType that is a PrefixedType that is an
       EncodingPrefixedType where the Type in the EncodingPrefixedType
       is one of (1) to (5).

The effect of this condition is to force the LIST encoding
instruction to be textually co-located with the SequenceOfType or
TypeWithConstraint to which it applies.

   Aside: This makes it clear to a reader that the encoding
   instruction applies to every use of the type no matter how it
   might be referenced.

The SequenceOfType in case (1) and the TypeWithConstraint in case (2)
are said to be "subject to" the LIST encoding instruction.

A SequenceOfType or TypeWithConstraint SHALL NOT be subject to more
than one LIST encoding instruction.

The base type of the component type of a SequenceOfType or
TypeWithConstraint that is subject to a LIST encoding instruction
MUST be one of the following:

(1) the BOOLEAN, INTEGER, ENUMERATED, REAL, OBJECT IDENTIFIER,
    RELATIVE-OID, GeneralizedTime, or UTCTime type, or

(2) the NCName, AnyURI, Name, or QName type from the
    AdditionalBasicDefinitions module [RXER].

   Aside: While it would be feasible to allow the component type to
   also be any character string type that is constrained such that
   all its abstract values have a length greater than zero and none
   of its abstract values contain any white space characters, testing
   whether this condition is satisfied can be quite involved. For
   the sake of simplicity, only certain immediately useful
   constrained UTF8String types, which are known to be suitable, are
   permitted (i.e., NCName, AnyURI, and Name).

The NamedType in a SequenceOfType or TypeWithConstraint that is
subject to a LIST encoding instruction MUST NOT be subject to an
ATTRIBUTE, ATTRIBUTE-REF, COMPONENT-REF, GROUP, ELEMENT-REF,
REF-AS-ELEMENT, SIMPLE-CONTENT, or TYPE-AS-VERSION encoding
instruction.

Example

   UpdateTimes ::= [LIST] SEQUENCE OF updateTime GeneralizedTime

13. The NAME Encoding Instruction

   The NAME encoding instruction causes an RXER encoder to use a
   nominated character string instead of a component's identifier
   wherever that identifier would otherwise appear in the encoding
   (e.g., as an element or attribute name).

   The notation for a NAME encoding instruction is defined as follows:

      NameInstruction ::= "NAME" "AS"? NCNameValue

   Example

      CHOICE {
          foo-att   [ATTRIBUTE] [NAME AS "Foo"] INTEGER,
          foo-elem [NAME "Foo"] INTEGER
      }

14. The REF-AS-ELEMENT Encoding Instruction

   The REF-AS-ELEMENT encoding instruction causes an RXER encoder to
   encode a value of the component to which it is applied as an element
   where the element's name is the name of the external DTD subset
   element type declaration referenced by the encoding instruction. In
   addition, the REF-AS-ELEMENT encoding instruction causes values of
   the Markup ASN.1 type to be restricted to conform to the content and
   attributes permitted by that element type declaration and its
   associated attribute-list declarations.

   The notation for a REF-AS-ELEMENT encoding instruction is defined as
   follows:

      RefAsElementInstruction ::=
          "REF-AS-ELEMENT" NameValue Namespace ? RefParameters

      Namespace ::= "NAMESPACE" AnyURIValue

   Taken together, the NameValue and the ContextParameter in the
   RefParameters (if present) MUST reference an element type declaration
   in an external DTD subset that is conformant with Namespaces in XML
   1.0 [XMLNS10].

   The Namespace is present if and only if the Name of the referenced
   element type declaration conforms to a PrefixedName (a QName)
   [XMLNS10], in which case the Namespace specifies the namespace name
   to be associated with the Prefix of the PrefixedName.

The referenced element type declaration MUST NOT require the presence
of attributes of type ENTITY or ENTITIES.

   Aside: Entity declarations are not supported by CRXER.

Example

   Suppose that the following external DTD subset has been defined
   with a system identifier of "http://www.example.com/inventory":

```
<?xml version='1.0'?>
<!ELEMENT product EMPTY>
<!ATTLIST product
     name       CDATA #IMPLIED
     partNumber CDATA #REQUIRED
     quantity   CDATA #REQUIRED >
```

   The product element type declaration can be referenced as an
   element in an ASN.1 type definition:

```
CHOICE {
    product  [REF-AS-ELEMENT "product"
                 CONTEXT "http://www.example.com/inventory"]
             Markup
}
```

   Here is the ASN.X translation of this ASN.1 type definition:

```
<type>
 <choice>
  <element elementType="product"
           context="http://www.example.com/inventory"/>
 </choice>
</type>
```

The identifier of a NamedType subject to a REF-AS-ELEMENT encoding
instruction does not contribute to the name of an element in an RXER
encoding. For the sake of consistency, the identifier SHOULD, where
possible, be the same as the Name of the referenced element type
declaration (or the LocalPart if the Name conforms to a
PrefixedName).

15. The REF-AS-TYPE Encoding Instruction

The REF-AS-TYPE encoding instruction causes values of the Markup
ASN.1 type to be restricted to conform to the content and attributes
permitted by a nominated element type declaration and its associated
attribute-list declarations in an external DTD subset.

The notation for a REF-AS-TYPE encoding instruction is defined as
follows:

    RefAsTypeInstruction ::= "REF-AS-TYPE" NameValue RefParameters

Taken together, the NameValue and the ContextParameter of the
RefParameters (if present) MUST reference an element type declaration
in an external DTD subset that is conformant with Namespaces in XML
1.0 [XMLNS10].

The referenced element type declaration MUST NOT require the presence
of attributes of type ENTITY or ENTITIES.

    Aside: Entity declarations are not supported by CRXER.

Example

    The product element type declaration can be referenced as a type
    in an ASN.1 definition:

        SEQUENCE OF
            inventoryItem
                [REF-AS-TYPE "product"
                    CONTEXT "http://www.example.com/inventory"]
                Markup

    Here is the ASN.X translation of this definition:

        <sequenceOf>
         <element name="inventoryItem">
          <type elementType="product"
                context="http://www.example.com/inventory"/>
         </element>
        </sequenceOf>

    Note that when an element type declaration is referenced as a
    type, the Name of the element type declaration does not contribute
    to RXER encodings. For example, child elements in the RXER
    encoding of values of the above SEQUENCE OF type would resemble
    the following:

        <inventoryItem name="hammer" partNumber="1543" quantity="29"/>

16.   The SCHEMA-IDENTITY Encoding Instruction

   The SCHEMA-IDENTITY encoding instruction associates a unique
   identifier, a URI [URI], with the ASN.1 module containing the
   encoding instruction. This encoding instruction has no effect on an
   RXER encoder but does have an effect on the translation of an ASN.1
   specification into an ASN.X representation.

   The notation for a SCHEMA-IDENTITY encoding instruction is defined as
   follows:

      SchemaIdentityInstruction ::= "SCHEMA-IDENTITY" AnyURIValue

   The character string specified by the AnyURIValue of each
   SCHEMA-IDENTITY encoding instruction MUST be distinct. In
   particular, successive versions of an ASN.1 module must each have a
   different schema identity URI value.

17.   The SIMPLE-CONTENT Encoding Instruction

   The SIMPLE-CONTENT encoding instruction causes an RXER encoder to
   encode a value of a component of a SEQUENCE or SET type without
   encapsulation in a child element.

   The notation for a SIMPLE-CONTENT encoding instruction is defined as
   follows:

      SimpleContentInstruction ::= "SIMPLE-CONTENT"

   A NamedType subject to a SIMPLE-CONTENT encoding instruction SHALL be
   in a ComponentType in a ComponentTypeList in a RootComponentTypeList.
   At most one such NamedType of a SEQUENCE or SET type is permitted to
   be subject to a SIMPLE-CONTENT encoding instruction. If any
   component is subject to a SIMPLE-CONTENT encoding instruction, then
   all other components in the same SEQUENCE or SET type definition MUST
   be attribute components. These tests are applied after the
   COMPONENTS OF transformation specified in X.680, Clause 24.4 [X.680].

      Aside: Child elements and simple content are mutually exclusive.
      Specification writers should note that use of the SIMPLE-CONTENT
      encoding instruction on a component of an extensible SEQUENCE or
      SET type means that all future extensions to the SEQUENCE or SET
      type are restricted to being attribute components with the limited
      set of types that are permitted for attribute components. Using
      an ATTRIBUTE encoding instruction instead of a SIMPLE-CONTENT
      encoding instruction avoids this limitation.

The base type of the type of a NamedType that is subject to a
SIMPLE-CONTENT encoding instruction SHALL NOT be:

(1) a SET or SET OF type, or

(2) a CHOICE type where the ChoiceType is not subject to a UNION
    encoding instruction, or

(3) a SEQUENCE type other than the one defining the QName type from
    the AdditionalBasicDefinitions module [RXER] (i.e., QName is
    allowed), or

(4) a SEQUENCE OF type where the SequenceOfType is not subject to a
    LIST encoding instruction, or

(5) an open type.

If the type of a NamedType subject to a SIMPLE-CONTENT encoding
instruction has abstract values with an empty character data
translation [RXER] (i.e., an empty encoding), then the NamedType
SHALL NOT be marked OPTIONAL or DEFAULT.

Example

```
SEQUENCE {
    units   [ATTRIBUTE] UTF8String,
    amount  [SIMPLE-CONTENT] INTEGER
}
```

18. The TARGET-NAMESPACE Encoding Instruction

The TARGET-NAMESPACE encoding instruction associates an XML namespace
name [XMLNS10], a URI [URI], with the type, object class, value,
object, and object set references defined in the ASN.1 module
containing the encoding instruction. In addition, it associates the
namespace name with each top-level NamedType in the RXER encoding
control section.

The notation for a TARGET-NAMESPACE encoding instruction is defined
as follows:

```
TargetNamespaceInstruction ::=
    "TARGET-NAMESPACE" AnyURIValue Prefix ?

Prefix ::= "PREFIX" NCNameValue
```

The AnyURIValue SHALL NOT specify an empty string.

Definition (target namespace): If an ASN.1 module contains a
TARGET-NAMESPACE encoding instruction, then the target namespace of
the module is the character string specified by the AnyURIValue of
the TARGET-NAMESPACE encoding instruction; otherwise, the target
namespace of the module is said to be absent.

Two or more ASN.1 modules MAY have the same non-absent target
namespace if and only if the expanded names of the top-level
attribute components are distinct across all those modules, the
expanded names of the top-level element components are distinct
across all those modules, and the defined type, object class, value,
object, and object set references are distinct in their category
across all those modules.

The Prefix, if present, suggests an NCName to use as the namespace
prefix in namespace declarations involving the target namespace. An
RXER encoder is not obligated to use the nominated namespace prefix.

If there are no top-level components, then the RXER encodings
produced using a module with a TARGET-NAMESPACE encoding instruction
are backward compatible with the RXER encodings produced by the same
module without the TARGET-NAMESPACE encoding instruction.

19. The TYPE-AS-VERSION Encoding Instruction

The TYPE-AS-VERSION encoding instruction causes an RXER encoder to
include an xsi:type attribute in the encoding of a value of the
component to which the encoding instruction is applied. This
attribute allows an XML Schema [XSD1] validator to select, if
available, the appropriate XML Schema translation for the version of
the ASN.1 specification used to create the encoding.

   Aside: Translations of an ASN.1 specification into a compatible
   XML Schema are expected to be slightly different across versions
   because of progressive extensions to the ASN.1 specification. Any
   incompatibilities between these translations can be accommodated
   if each version uses a different target namespace. The target
   namespace will be evident in the value of the xsi:type attribute
   and will cause an XML Schema validator to use the appropriate
   version. This mechanism also accommodates an ASN.1 type that is
   renamed in a later version of the ASN.1 specification.

The notation for a TYPE-AS-VERSION encoding instruction is defined as
follows:

    TypeAsVersionInstruction ::= "TYPE-AS-VERSION"

The Type in a NamedType that is subject to a TYPE-AS-VERSION encoding
instruction MUST be a namespace-qualified reference [RXER].

The addition of a TYPE-AS-VERSION encoding instruction does not
affect the backward compatibility of RXER encodings.

> Aside: In a translation of an ASN.1 specification into XML Schema,
> any Type in a NamedType that is subject to a TYPE-AS-VERSION
> encoding instruction is expected to be translated into the
> XML Schema anyType so that the xsi:type attribute acts as a switch
> to select the appropriate version.

20. The TYPE-REF Encoding Instruction

The TYPE-REF encoding instruction causes values of the Markup ASN.1
type to be restricted to conform to a specific XML Schema named type,
RELAX NG named pattern or an ASN.1 defined type.

> Aside: Referencing an ASN.1 type in a TYPE-REF encoding
> instruction does not have the effect of imposing a requirement to
> preserve the Infoset [INFOSET] representation of the RXER encoding
> of an abstract value of the type. It is still sufficient to
> preserve just the abstract value.

The notation for a TYPE-REF encoding instruction is defined as
follows:

    TypeRefInstruction ::= "TYPE-REF" QNameValue RefParameters

Taken together, the QNameValue and the ContextParameter of the
RefParameters (if present) MUST reference an XML Schema named type, a
RELAX NG named pattern, or an ASN.1 defined type.

A referenced XML Schema type MUST NOT require the presence of values
for the XML Schema ENTITY or ENTITIES types.

> Aside: Entity declarations are not supported by CRXER.

The QNameValue SHALL NOT be a direct reference to the XML Schema
NOTATION type [XSD2] (i.e., the namespace name
"http://www.w3.org/2001/XMLSchema" and local name "NOTATION");
however, a reference to an XML Schema type derived from the NOTATION
type is permitted.

> Aside: This restriction is to ensure that the lexical space [XSD2]
> of the referenced type is actually populated with the names of
> notations [XSD1].

Example

```
MyDecimal ::=
    [TYPE-REF {
        namespace-name "http://www.w3.org/2001/XMLSchema",
        local-name      "decimal" }]
    Markup
```

Note that the ASN.X translation of this ASN.1 type definition
provides a more natural way to reference the XML Schema decimal
type:

```
<namedType xmlns:xs="http://www.w3.org/2001/XMLSchema"
           name="MyDecimal">
 <type ref="xs:decimal" embedded="true"/>
</namedType>
```

21. The UNION Encoding Instruction

The UNION encoding instruction causes an RXER encoder to encode the
value of an alternative of a CHOICE type without encapsulation in a
child element. The chosen alternative is optionally indicated with a
member attribute. The optional PrecedenceList also allows a
specification writer to alter the order in which an RXER decoder will
consider the alternatives of the CHOICE as it determines which
alternative has been used (if the actual alternative has not been
specified through the member attribute).

The notation for a UNION encoding instruction is defined as follows:

    UnionInstruction ::= "UNION" AlternativesPrecedence ?

    AlternativesPrecedence ::= "PRECEDENCE" PrecedenceList

    PrecedenceList ::= identifier PrecedenceList ?

The Type in the EncodingPrefixedType for a UNION encoding instruction
SHALL be either:

(1) a BuiltinType that is a ChoiceType, or

(2) a ConstrainedType that is not a TypeWithConstraint where the Type
    in the ConstrainedType is one of (1) to (4), or

(3) a BuiltinType that is a PrefixedType that is a TaggedType where
    the Type in the TaggedType is one of (1) to (4), or

   (4) a BuiltinType that is a PrefixedType that is an
       EncodingPrefixedType where the Type in the EncodingPrefixedType
       is one of (1) to (4).

   The ChoiceType in case (1) is said to be "subject to" the UNION
   encoding instruction.

   The base type of the type of each alternative of a ChoiceType that is
   subject to a UNION encoding instruction SHALL NOT be:

   (1) a CHOICE, SET, or SET OF type, or

   (2) a SEQUENCE type other than the one defining the QName type from
       the AdditionalBasicDefinitions module [RXER] (i.e., QName is
       allowed), or

   (3) a SEQUENCE OF type where the SequenceOfType is not subject to a
       LIST encoding instruction, or

   (4) an open type.

   Each identifier in the PrecedenceList MUST be the identifier of a
   NamedType in the ChoiceType.

   A particular identifier SHALL NOT appear more than once in the same
   PrecedenceList.

   Every NamedType in a ChoiceType that is subject to a UNION encoding
   instruction MUST NOT be subject to an ATTRIBUTE, ATTRIBUTE-REF,
   COMPONENT-REF, GROUP, ELEMENT-REF, REF-AS-ELEMENT, SIMPLE-CONTENT, or
   TYPE-AS-VERSION encoding instruction.

   Example

      [UNION PRECEDENCE basicName] CHOICE {
          extendedName UTF8String,
          basicName   PrintableString
      }

22. The VALUES Encoding Instruction

   The VALUES encoding instruction causes an RXER encoder to use
   nominated names instead of the identifiers that would otherwise
   appear in the encoding of a value of a BIT STRING, ENUMERATED, or
   INTEGER type.

The notation for a VALUES encoding instruction is defined as follows:

```
ValuesInstruction ::=
    "VALUES" AllValuesMapped ? ValueMappingList ?

AllValuesMapped ::= AllCapitalized | AllUppercased

AllCapitalized ::= "ALL" "CAPITALIZED"

AllUppercased ::= "ALL" "UPPERCASED"

ValueMappingList ::= ValueMapping ValueMappingList ?

ValueMapping ::= "," identifier "AS" NCNameValue
```

The Type in the EncodingPrefixedType for a VALUES encoding
instruction SHALL be either:

(1) a BuiltinType that is a BitStringType with a NamedBitList, or

(2) a BuiltinType that is an EnumeratedType, or

(3) a BuiltinType that is an IntegerType with a NamedNumberList, or

(4) a ConstrainedType that is not a TypeWithConstraint where the Type
    in the ConstrainedType is one of (1) to (6), or

(5) a BuiltinType that is a PrefixedType that is a TaggedType where
    the Type in the TaggedType is one of (1) to (6), or

(6) a BuiltinType that is a PrefixedType that is an
    EncodingPrefixedType where the Type in the EncodingPrefixedType
    is one of (1) to (6).

The effect of this condition is to force the VALUES encoding
instruction to be textually co-located with the type definition to
which it applies.

The BitStringType, EnumeratedType, or IntegerType in case (1), (2),
or (3), respectively, is said to be "subject to" the VALUES encoding
instruction.

A BitStringType, EnumeratedType, or IntegerType SHALL NOT be subject
to more than one VALUES encoding instruction.

Each identifier in a ValueMapping MUST be an identifier appearing in
the NamedBitList, Enumerations, or NamedNumberList, as the case may
be.

The identifier in a ValueMapping SHALL NOT be the same as the
identifier in any other ValueMapping for the same ValueMappingList.

Definition (replacement name): Each identifier in a BitStringType,
EnumeratedType, or IntegerType subject to a VALUES encoding
instruction has a replacement name. If there is a ValueMapping for
the identifier, then the replacement name is the character string
specified by the NCNameValue in the ValueMapping; else if
AllCapitalized is used, then the replacement name is the identifier
with the first character uppercased; else if AllUppercased is used,
then the replacement name is the identifier with all its characters
uppercased; otherwise, the replacement name is the identifier.

The replacement names for the identifiers in a BitStringType subject
to a VALUES encoding instruction MUST be distinct.

The replacement names for the identifiers in an EnumeratedType
subject to a VALUES encoding instruction MUST be distinct.

The replacement names for the identifiers in an IntegerType subject
to a VALUES encoding instruction MUST be distinct.

Example

```
   Traffic-Light ::= [VALUES ALL CAPITALIZED, red AS "RED"]
       ENUMERATED {
           red,    -- Replacement name is RED.
           amber,  -- Replacement name is Amber.
           green   -- Replacement name is Green.
       }
```

23. Insertion Encoding Instructions

Certain of the RXER encoding instructions are categorized as
insertion encoding instructions. The insertion encoding instructions
are the NO-INSERTIONS, HOLLOW-INSERTIONS, SINGULAR-INSERTIONS,
UNIFORM-INSERTIONS, and MULTIFORM-INSERTIONS encoding instructions
(whose notations are described respectively by
NoInsertionsInstruction, HollowInsertionsInstruction,
SingularInsertionsInstruction, UniformInsertionsInstruction, and
MultiformInsertionsInstruction).

The notation for the insertion encoding instructions is defined as
follows:

```
    InsertionsInstruction ::=
        NoInsertionsInstruction |
        HollowInsertionsInstruction |
        SingularInsertionsInstruction |
        UniformInsertionsInstruction |
        MultiformInsertionsInstruction

    NoInsertionsInstruction ::= "NO-INSERTIONS"

    HollowInsertionsInstruction ::= "HOLLOW-INSERTIONS"

    SingularInsertionsInstruction ::= "SINGULAR-INSERTIONS"

    UniformInsertionsInstruction ::= "UNIFORM-INSERTIONS"

    MultiformInsertionsInstruction ::= "MULTIFORM-INSERTIONS"
```

Using the GROUP encoding instruction on components with extensible
types can lead to situations where an unknown extension could be
associated with more than one extension insertion point. The
insertion encoding instructions remove this ambiguity by limiting the
form that extensions can take. That is, the insertion encoding
instructions indicate what extensions can be made to an ASN.1
specification without breaking forward compatibility for RXER
encodings.

    Aside: Forward compatibility means the ability for a decoder to
    successfully decode an encoding containing extensions introduced
    into a version of the specification that is more recent than the
    one used by the decoder.

In the most general case, an extension to a CHOICE, SET, or SEQUENCE
type will generate zero or more attributes and zero or more elements,
due to the potential use of the GROUP and ATTRIBUTE encoding
instructions by the extension.

The MULTIFORM-INSERTIONS encoding instruction indicates that the RXER
encodings produced by forward-compatible extensions to a type will
always consist of one or more elements and zero or more attributes.
No restriction is placed on the names of the elements.

    Aside: Of necessity, the names of the attributes will all be
    different in any given encoding.

The UNIFORM-INSERTIONS encoding instruction indicates that the RXER
encodings produced by forward-compatible extensions to a type will
always consist of one or more elements having the same expanded name,
and zero or more attributes. The expanded name shared by the

elements in one particular encoding is not required to be the same as
the expanded name shared by the elements in any other encoding of the
extension. For example, in one encoding of the extension the
elements might all be called "foo", while in another encoding of the
extension they might all be called "bar".

The SINGULAR-INSERTIONS encoding instruction indicates that the RXER
encodings produced by forward-compatible extensions to a type will
always consist of a single element and zero or more attributes. The
name of the single element is not required to be the same in every
possible encoding of the extension.

The HOLLOW-INSERTIONS encoding instruction indicates that the RXER
encodings produced by forward-compatible extensions to a type will
always consist of zero elements and zero or more attributes.

The NO-INSERTIONS encoding instruction indicates that no forward-
compatible extensions can be made to a type.

Examples of forward-compatible extensions are provided in Appendix C.

The Type in the EncodingPrefixedType for an insertion encoding
instruction SHALL be either:

(1) a BuiltinType that is a ChoiceType where the ChoiceType is not
    subject to a UNION encoding instruction, or

(2) a BuiltinType that is a SequenceType or SetType, or

(3) a ConstrainedType that is not a TypeWithConstraint where the Type
    in the ConstrainedType is one of (1) to (5), or

(4) a BuiltinType that is a PrefixedType that is a TaggedType where
    the Type in the TaggedType is one of (1) to (5), or

(5) a BuiltinType that is a PrefixedType that is an
    EncodingPrefixedType where the Type in the EncodingPrefixedType
    is one of (1) to (5).

Case (2) is not permitted when the insertion encoding instruction is
the SINGULAR-INSERTIONS, UNIFORM-INSERTIONS, or MULTIFORM-INSERTIONS
encoding instruction.

      Aside: Because extensions to a SET or SEQUENCE type are serial and
      effectively optional, the SINGULAR-INSERTIONS, UNIFORM-INSERTIONS,
      and MULTIFORM-INSERTIONS encoding instructions offer no advantage
      over unrestricted extensions (for a SET or SEQUENCE). For
      example, an optional series of singular insertions generates zero
      or more elements and zero or more attributes, just like an
      unrestricted extension.

   The Type in case (1) or case (2) is said to be "subject to" the
   insertion encoding instruction.

   The Type in case (1) or case (2) MUST be extensible, either
   explicitly or by default.

   A Type SHALL NOT be subject to more than one insertion encoding
   instruction.

   The insertion encoding instructions indicate what kinds of extensions
   can be made to a type without breaking forward compatibility, but
   they do not prohibit extensions that do break forward compatibility.
   That is, it is not an error for a type's base type to contain
   extensions that do not satisfy an insertion encoding instruction
   affecting the type. However, if any such extensions are made, then a
   new value SHOULD be introduced into the extensible set of permitted
   values for a version indicator attribute, or attributes (see
   Section 24), whose scope encompasses the extensions. An example is
   provided in Appendix C.

24. The VERSION-INDICATOR Encoding Instruction

   The VERSION-INDICATOR encoding instruction provides a mechanism for
   RXER decoders to be alerted that an encoding contains extensions that
   break forward compatibility (see the preceding section).

   The notation for a VERSION-INDICATOR encoding instruction is defined
   as follows:

      VersionIndicatorInstruction ::= "VERSION-INDICATOR"

   A NamedType that is subject to a VERSION-INDICATOR encoding
   instruction MUST also be subject to an ATTRIBUTE encoding
   instruction.

   The type of the NamedType that is subject to the VERSION-INDICATOR
   encoding instruction MUST be directly or indirectly a constrained
   type where the set of permitted values is defined to be extensible.
   Each value represents a different version of the ASN.1 specification.
   Ordinarily, an application will set the value of a version indicator

attribute to be the last of these permitted values. An application
MAY set the value of the version indicator attribute to the value
corresponding to an earlier version of the specification if it has
not used any of the extensions added in a subsequent version.

If an RXER decoder encounters a value of the type that is not one of
the root values or extension additions (but that is still allowed
since the set of permitted values is extensible), then this indicates
that the decoder is using a version of the ASN.1 specification that
is not compatible with the version used to produce the encoding. In
such cases, the decoder SHOULD treat the element containing the
attribute as having an unknown ASN.1 type.

   Aside: A version indicator attribute only indicates an
   incompatibility with respect to RXER encodings. Other encodings
   are not affected because the GROUP encoding instruction does not
   apply to them.

Examples

   In this first example, the decoder is using an incompatible older
   version if the value of the version attribute in a received RXER
   encoding is not 1, 2, or 3.

```
      SEQUENCE {
          version  [ATTRIBUTE] [VERSION-INDICATOR]
                      INTEGER (1, ..., 2..3),
          message  MessageType
      }
```

   In this second example, the decoder is using an incompatible older
   version if the value of the format attribute in a received RXER
   encoding is not "1.0", "1.1", or "2.0".

```
      SEQUENCE {
          format   [ATTRIBUTE] [VERSION-INDICATOR]
                      UTF8String ("1.0", ..., "1.1" | "2.0"),
          message  MessageType
      }
```

   An extensive example is provided in Appendix C.

It is not necessary for every extensible type to have its own version
indicator attribute. It would be typical for only the types of
top-level element components to include a version indicator
attribute, which would serve as the version indicator for all of the
nested components.

25. The GROUP Encoding Instruction

   The GROUP encoding instruction causes an RXER encoder to encode a
   value of the component to which it is applied without encapsulation
   as an element. It allows the construction of non-trivial content
   models for element content.

   The notation for a GROUP encoding instruction is defined as follows:

      GroupInstruction ::= "GROUP"

   The base type of the type of a NamedType that is subject to a GROUP
   encoding instruction SHALL be either:

   (1) a SEQUENCE, SET, or SET OF type, or

   (2) a CHOICE type where the ChoiceType is not subject to a UNION
       encoding instruction, or

   (3) a SEQUENCE OF type where the SequenceOfType is not subject to a
       LIST encoding instruction.

   The SEQUENCE type in case (1) SHALL NOT be the associated type for a
   built-in type, SHALL NOT be a type from the
   AdditionalBasicDefinitions module [RXER], and SHALL NOT contain a
   component that is subject to a SIMPLE-CONTENT encoding instruction.

      Aside: Thus, the CHARACTER STRING, EMBEDDED PDV, EXTERNAL, REAL,
      and QName types are excluded.

   The CHOICE type in case (2) SHALL NOT be a type from the
   AdditionalBasicDefinitions module.

      Aside: Thus, the Markup type is excluded.

   Definition (visible component): Ignoring all type constraints, the
   visible components for a type that is directly or indirectly a
   combining ASN.1 type (i.e., SEQUENCE, SET, CHOICE, SEQUENCE OF, or
   SET OF) is the set of components of the combining type definition
   plus, for each NamedType (of the combining type definition) that is
   subject to a GROUP encoding instruction, the visible components for
   the type of the NamedType. The visible components are determined
   after the COMPONENTS OF transformation specified in X.680, Clause
   24.4 [X.680].

   Aside: The set of visible attribute and element components for a
   type is the set of all the components of the type, and any nested
   types, that describe attributes and child elements appearing in
   the RXER encodings of values of the outer type.

A GROUP encoding instruction MUST NOT be used where it would cause a
NamedType to be a visible component of the type of that same
NamedType (which is only possible if the type definition is
recursive).

   Aside: Components subject to a GROUP encoding instruction might be
   translated into a compatible XML Schema [XSD1] as group
   definitions. A NamedType that is visible to its own type is
   analogous to a circular group, which XML Schema disallows.

Section 25.1 imposes additional conditions on the use of the GROUP
encoding instruction.

In any use of the GROUP encoding instruction, there is a type, the
including type, that contains the component subject to the GROUP
encoding instruction, and a type, the included type, that is the base
type of that component. Either type can have an extensible content
model, either by directly using ASN.1 extensibility or by including
through another GROUP encoding instruction some other type that is
extensible.

The including and included types may be defined in different ASN.1
modules, in which case the owner of the including type, i.e., the
person or organization having the authority to add extensions to the
including type's definition, may be different from the owner of the
included type.

If the owner of the including type is not using the most recent
version of the included type's definition, then the owner of the
including type might add an extension to the including type that is
valid with respect to the older version of the included type, but is
later found to be invalid when the latest versions of the including
and included type definitions are brought together (perhaps by a
third party). Although the owner of the including type must
necessarily be aware of the existence of the included type, the
reverse is not necessarily true. The owner of the included type
could add an extension to the included type without realizing that it
invalidates someone else's including type.

To avoid these problems, a GROUP encoding instruction MUST NOT be
used if:

(1) the included type is defined in a different module from the
    including type, and

(2) the included type has an extensible content model, and

(3) changes to the included type are not coordinated with the owner
    of the including type.

Changes in the included type are coordinated with the owner of the
including type if:

(1) the owner of the included type is also the owner of the including
    type, or

(2) the owner of the including type is collaborating with the owner
    of the included type, or

(3) all changes will be vetted by a common third party before being
    approved and published.

25.1. Unambiguous Encodings

Unregulated use of the GROUP encoding instruction can easily lead to
specifications in which distinct abstract values have
indistinguishable RXER encodings, i.e., ambiguous encodings. This
section imposes restrictions on the use of the GROUP encoding
instruction to ensure that distinct abstract values have distinct
RXER encodings. In addition, these restrictions ensure that an
abstract value can be easily decoded in a single pass without
back-tracking.

An RXER decoder for an ASN.1 type can be abstracted as a recognizer
for a notional language, consisting of element and attribute expanded
names, where the type definition describes the grammar for that
language (in fact it is a context-free grammar). The restrictions on
a type definition to ensure easy, unambiguous decoding are more
conveniently, completely, and simply expressed as conditions on this
associated grammar. Implementations are not expected to verify type
definitions exactly in the manner to be described; however, the
procedure used MUST produce the same result.

Section 25.1.1 describes the procedure for recasting as a grammar a
type definition containing components subject to the GROUP encoding
instruction. Sections 25.1.2 and 25.1.3 specify conditions that the

grammar must satisfy for the type definition to be valid. Section
25.1.4 describes how unrecognized attributes are accepted by the
grammar for an extensible type.

Appendices A and B have extensive examples.

25.1.1. Grammar Construction

A grammar consists of a collection of productions. A production has
a left-hand side and a right-hand side (in this document, separated
by the "::=" symbol). The left-hand side (in a context-free grammar)
is a single non-terminal symbol. The right-hand side is a sequence
of non-terminal and terminal symbols. The terminal symbols are the
lexical items of the language that the grammar describes. One of the
non-terminals is nominated to be the start symbol. A valid sequence
of terminals for the language can be generated from the grammar by
beginning with the start symbol and repeatedly replacing any
non-terminal with the right-hand side of one of the productions where
that non-terminal is on the production's left-hand side. The final
sequence of terminals is achieved when there are no remaining
non-terminals to replace.

    Aside: X.680 describes the ASN.1 basic notation using a
    context-free grammar.

Each NamedType has an associated primary and secondary non-terminal.

    Aside: The secondary non-terminal for a NamedType is used when the
    base type of the type in the NamedType is a SEQUENCE OF type or
    SET OF type.

Each ExtensionAddition and ExtensionAdditionAlternative has an
associated non-terminal. There is a non-terminal associated with the
extension insertion point of each extensible type. There is also a
primary start non-terminal (this is the start symbol) and a secondary
start non-terminal. The exact nature of the non-terminals is not
important, however all the non-terminals MUST be mutually distinct.

It is adequate for most of the examples in this document (though not
in the most general case) for the primary non-terminal for a
NamedType to be the identifier of the NamedType, for the primary
start non-terminal to be S, for the non-terminals for the instances
of ExtensionAddition and ExtensionAdditionAlternative to be E1, E2,
E3, and so on, and for the non-terminals for the extension insertion
points to be I1, I2, I3, and so on. The secondary non-terminals are
labelled by appending a "'" character to the primary non-terminal
label, e.g., the primary and secondary start non-terminals are S and
S', respectively.

Each NamedType and extension insertion point has an associated
terminal. There exists a terminal called the general extension
terminal that is not associated with any specific notation. The
general extension terminal and the terminals for the extension
insertion points are used to represent elements in unknown
extensions. The exact nature of the terminals is not important;
however, the aforementioned terminals MUST be mutually distinct. The
terminals are further categorized as either element terminals or
attribute terminals. A terminal for a NamedType is an attribute
terminal if its associated NamedType is an attribute component;
otherwise, it is an element terminal. The general extension terminal
and the terminals for the extension insertion points are categorized
as element terminals.

Terminals for attributes in unknown extensions are not explicitly
provided in the grammar. Certain productions in the grammar are
categorized as insertion point productions, and their role in
accepting unknown attributes is described in Section 25.1.4.

In the examples in this document, the terminal for a component other
than an attribute component will be represented as the local name of
the expanded name of the component enclosed in double quotes, and the
terminal for an attribute component will be represented as the local
name of the expanded name of the component prefixed by the '@'
character and enclosed in double quotes. The general extension
terminal will be represented as "*" and the terminals for the
extension insertion points will be represented as "*1", "*2", "*3",
and so on.

The productions generated from a NamedType depend on the base type of
the type of the NamedType. The productions for the start
non-terminals depend on the combining type definition being tested.
In either case, the procedure for generating productions takes a
primary non-terminal, a secondary non-terminal (sometimes), and a
type definition.

The grammar is constructed beginning with the start non-terminals and
the combining type definition being tested.

A grammar is constructed after the COMPONENTS OF transformation
specified in X.680, Clause 24.4 [X.680].

Given a primary non-terminal, N, and a type where the base type is a
SEQUENCE or SET type, a production is added to the grammar with N as
the left-hand side. The right-hand side is constructed from an
initial empty state according to the following cases considered in
order:

   (1) If an initial RootComponentTypeList is present in the base type,
       then the sequence of primary non-terminals for the components
       nested in that RootComponentTypeList are appended to the right-
       hand side in the order of their definition.

   (2) If an ExtensionAdditions instance is present in the base type and
       not empty, then the non-terminal for the first ExtensionAddition
       nested in the ExtensionAdditions instance is appended to the
       right-hand side.

   (3) If an ExtensionAdditions instance is empty or not present in the
       base type, and the base type is extensible (explicitly or by
       default), and the base type is not subject to a NO-INSERTIONS or
       HOLLOW-INSERTIONS encoding instruction, then the non-terminal for
       the extension insertion point of the base type is appended to the
       right-hand side.

   (4) If a final RootComponentTypeList is present in the base type,
       then the primary non-terminals for the components nested in that
       RootComponentTypeList are appended to the right-hand side in the
       order of their definition.

   The production is an insertion point production if an
   ExtensionAdditions instance is empty or not present in the base type,
   and the base type is extensible (explicitly or by default), and the
   base type is not subject to a NO-INSERTIONS encoding instruction.

   If a component in a ComponentTypeList (in either a
   RootComponentTypeList or an ExtensionAdditionGroup) is marked
   OPTIONAL or DEFAULT, then a production with the primary non-terminal
   of the component as the left-hand side and an empty right-hand side
   is added to the grammar.

   If a component (regardless of the ASN.1 combining type containing it)
   is subject to a GROUP encoding instruction, then one or more
   productions constructed according to the component's type are added
   to the grammar. Each of these productions has the primary
   non-terminal of the component as the left-hand side.

   If a component (regardless of the ASN.1 combining type containing it)
   is not subject to a GROUP encoding instruction, then a production is
   added to the grammar with the primary non-terminal of the component
   as the left-hand side and the terminal of the component as the
   right-hand side.

Example

   Consider the following ASN.1 type definition:

       SEQUENCE {
           -- Start of initial RootComponentTypeList.
           one   [ATTRIBUTE] UTF8String,
           two   BOOLEAN OPTIONAL,
           three   INTEGER
           -- End of initial RootComponentTypeList.
       }

   Here is the grammar derived from this type:

       S ::= one two three
       one ::= "@one"
       two ::= "two"
       two ::=
       three ::= "three"

For each ExtensionAddition (of a SEQUENCE or SET base type), a
production is added to the grammar where the left-hand side is the
non-terminal for the ExtensionAddition and the right-hand side is
initially empty. If the ExtensionAddition is a ComponentType, then
the primary non-terminal for the NamedType in the ComponentType is
appended to the right-hand side; otherwise (an
ExtensionAdditionGroup), the sequence of primary non-terminals for
the components nested in the ComponentTypeList in the
ExtensionAdditionGroup are appended to the right-hand side in the
order of their definition. If the ExtensionAddition is followed by
another ExtensionAddition, then the non-terminal for the next
ExtensionAddition is appended to the right-hand side; otherwise, if
the base type is not subject to a NO-INSERTIONS or HOLLOW-INSERTIONS
encoding instruction, then the non-terminal for the extension
insertion point of the base type is appended to the right-hand side.
If the ExtensionAddition is not followed by another ExtensionAddition
and the base type is not subject to a NO-INSERTIONS encoding
instruction, then the production is an insertion point production.
If the empty sequence of terminals cannot be generated from the
production (it may be necessary to wait until the grammar is
otherwise complete before making this determination), then another
production is added to the grammar where the left-hand side is the
non-terminal for the ExtensionAddition and the right-hand side is
empty.

   Aside: An extension is always effectively optional since a sender
   may be using an earlier version of the ASN.1 specification where
   none, or only some, of the extensions have been defined.

   Aside: The grammar generated for ExtensionAdditions is structured
   to take account of the condition that an extension can only be
   used if all the earlier extensions are also used [X.680].

If a SEQUENCE or SET base type is extensible (explicitly or by
default) and is not subject to a NO-INSERTIONS or HOLLOW-INSERTIONS
encoding instruction, then:

(1) a production is added to the grammar where the left-hand side is
    the non-terminal for the extension insertion point of the base
    type and the right-hand side is the general extension terminal
    followed by the non-terminal for the extension insertion point,
    and

(2) a production is added to the grammar where the left-hand side is
    the non-terminal for the extension insertion point and the
    right-hand side is empty.

Example

   Consider the following ASN.1 type definition:

```
SEQUENCE {
    -- Start of initial RootComponentTypeList.
    one     BOOLEAN,
    two     INTEGER OPTIONAL,
    -- End of initial RootComponentTypeList.
    ...,
    -- Start of ExtensionAdditions.
    four  INTEGER,  -- First ExtensionAddition (E1).
    five  BOOLEAN OPTIONAL,  -- Second ExtensionAddition (E2).
    [[ -- An ExtensionAdditionGroup.
        six     UTF8String,
        seven   INTEGER OPTIONAL
    ]], -- Third ExtensionAddition (E3).
    -- End of ExtensionAdditions.
    -- The extension insertion point is here (I1).
    ...,
    -- Start of final RootComponentTypeList.
    three  INTEGER
}
```

   Here is the grammar derived from this type:

```
S ::= one two E1 three

E1 ::= four E2
E1 ::=
```

```
        E2 ::= five E3
        E3 ::= six seven I1
        E3 ::=

        I1 ::= "*" I1
        I1 ::=

        one ::= "one"
        two ::= "two"
        two ::=
        three ::= "three"
        four ::= "four"
        five ::= "five"
        five ::=
        six ::= "six"
        seven ::= "seven"
        seven ::=
```

   If the SEQUENCE type were subject to a NO-INSERTIONS or
   HOLLOW-INSERTIONS encoding instruction, then the productions for
   I1 would not appear, and the first production for E3 would be:

```
        E3 ::= six seven
```

Given a primary non-terminal, N, and a type where the base type is a
CHOICE type:

(1) A production is added to the grammar for each NamedType nested in
    the RootAlternativeTypeList of the base type, where the left-hand
    side is N and the right-hand side is the primary non-terminal for
    the NamedType.

(2) A production is added to the grammar for each
    ExtensionAdditionAlternative of the base type, where the left-
    hand side is N and the right-hand side is the non-terminal for
    the ExtensionAdditionAlternative.

(3) If the base type is extensible (explicitly or by default) and the
    base type is not subject to an insertion encoding instruction,
    then:

    (a) A production is added to the grammar where the left-hand side
        is N and the right-hand side is the non-terminal for the
        extension insertion point of the base type. This production
        is an insertion point production.

(b) A production is added to the grammar where the left-hand side
    is the non-terminal for the extension insertion point of the
    base type and the right-hand side is the general extension
    terminal followed by the non-terminal for the extension
    insertion point.

(c) A production is added to the grammar where the left-hand side
    is the non-terminal for the extension insertion point of the
    base type and the right-hand side is empty.

(4) If the base type is subject to a HOLLOW-INSERTIONS encoding
    instruction, then a production is added to the grammar where the
    left-hand side is N and the right-hand side is empty. This
    production is an insertion point production.

(5) If the base type is subject to a SINGULAR-INSERTIONS encoding
    instruction, then a production is added to the grammar where the
    left-hand side is N and the right-hand side is the general
    extension terminal. This production is an insertion point
    production.

(6) If the base type is subject to a UNIFORM-INSERTIONS encoding
    instruction, then:

    (a) A production is added to the grammar where the left-hand side
        is N and the right-hand side is the general extension
        terminal.

           Aside: This production is used to verify the correctness
           of an ASN.1 type definition, but would not be used in the
           implementation of an RXER decoder. The next production
           takes precedence over it for accepting an unknown element.

    (b) A production is added to the grammar where the left-hand side
        is N and the right-hand side is the terminal for the
        extension insertion point of the base type followed by the
        non-terminal for the extension insertion point. This
        production is an insertion point production.

    (c) A production is added to the grammar where the left-hand side
        is the non-terminal for the extension insertion point of the
        base type and the right-hand side is the terminal for the
        extension insertion point followed by the non-terminal for
        the extension insertion point.

    (d) A production is added to the grammar where the left-hand side
        is the non-terminal for the extension insertion point of the
        base type and the right-hand side is empty.

(7) If the base type is subject to a MULTIFORM-INSERTIONS encoding
    instruction, then:

    (a) A production is added to the grammar where the left-hand side
        is N and the right-hand side is the general extension
        terminal followed by the non-terminal for the extension
        insertion point of the base type. This production is an
        insertion point production.

    (b) A production is added to the grammar where the left-hand side
        is the non-terminal for the extension insertion point of the
        base type and the right-hand side is the general extension
        terminal followed by the non-terminal for the extension
        insertion point.

    (c) A production is added to the grammar where the left-hand side
        is the non-terminal for the extension insertion point of the
        base type and the right-hand side is empty.

If an ExtensionAdditionAlternative is a NamedType, then a production
is added to the grammar where the left-hand side is the non-terminal
for the ExtensionAdditionAlternative and the right-hand side is the
primary non-terminal for the NamedType.

If an ExtensionAdditionAlternative is an
ExtensionAdditionAlternativesGroup, then a production is added to the
grammar for each NamedType nested in the
ExtensionAdditionAlternativesGroup, where the left-hand side is the
non-terminal for the ExtensionAdditionAlternative and the right-hand
side is the primary non-terminal for the NamedType.

   Example

      Consider the following ASN.1 type definition:

         CHOICE {
             -- Start of RootAlternativeTypeList.
             one   BOOLEAN,
             two   INTEGER,
             -- End of RootAlternativeTypeList.
             ...,
             -- Start of ExtensionAdditionAlternatives.
             three INTEGER, -- First ExtensionAdditionAlternative (E1).
             [[-- An ExtensionAdditionAlternativesGroup.
                four UTF8String,
                five INTEGER
             ]] -- Second ExtensionAdditionAlternative (E2).
             -- The extension insertion point is here (I1).
         }

      Here is the grammar derived from this type:

         S ::= one
         S ::= two
         S ::= E1
         S ::= E2
         S ::= I1

         I1 ::= "*" I1
         I1 ::=

         E1 ::= three
         E2 ::= four
         E2 ::= five

         one ::= "one"
         two ::= "two"
         three ::= "three"
         four ::= "four"
         five ::= "five"

      If the CHOICE type were subject to a NO-INSERTIONS encoding
      instruction, then the fifth, sixth, and seventh productions would
      be removed.

      If the CHOICE type were subject to a HOLLOW-INSERTIONS encoding
      instruction, then the fifth, sixth, and seventh productions would
      be replaced by:

```
S ::=
```

If the CHOICE type were subject to a SINGULAR-INSERTIONS encoding
instruction, then the fifth, sixth, and seventh productions would
be replaced by:

```
S ::= "*"
```

If the CHOICE type were subject to a UNIFORM-INSERTIONS encoding
instruction, then the fifth and sixth productions would be
replaced by:

```
S ::= "*"
S ::= "*1" I1

I1 ::= "*1" I1
```

If the CHOICE type were subject to a MULTIFORM-INSERTIONS encoding
instruction, then the fifth production would be replaced by:

```
S ::= "*" I1
```

Constraints on a SEQUENCE, SET, or CHOICE type are ignored. They do
not affect the grammar being generated.

   Aside: This avoids an awkward situation where values of a subtype
   have to be decoded differently from values of the parent type. It
   also simplifies the verification procedure.

Given a primary non-terminal, N, and a type that has a SEQUENCE OF or
SET OF base type and that permits a value of size zero (i.e., an
empty sequence or set):

(1) a production is added to the grammar where the left-hand side of
    the production is N and the right-hand side is the primary
    non-terminal for the NamedType of the component of the
    SEQUENCE OF or SET OF base type, followed by N, and

(2) a production is added to the grammar where the left-hand side of
    the production is N and the right-hand side is empty.

Given a primary non-terminal, N, a secondary non-terminal, N', and a
type that has a SEQUENCE OF or SET OF base type and that does not
permit a value of size zero:

(1) a production is added to the grammar where the left-hand side of
    the production is N and the right-hand side is the primary
    non-terminal for the NamedType of the component of the
    SEQUENCE OF or SET OF base type, followed by N', and

(2) a production is added to the grammar where the left-hand side of
    the production is N' and the right-hand side is the primary
    non-terminal for the NamedType of the component of the
    SEQUENCE OF or SET OF base type, followed by N', and

(3) a production is added to the grammar where the left-hand side of
    the production is N' and the right-hand side is empty.

Example

    Consider the following ASN.1 type definition:

        SEQUENCE SIZE(1..MAX) OF number INTEGER

    Here is the grammar derived from this type:

        S ::= number S'
        S' ::= number S'
        S' ::=

        number ::= "number"

All inner subtyping (InnerTypeContraints) is ignored for the purposes
of deciding whether a value of size zero is permitted by a
SEQUENCE OF or SET OF type.

This completes the description of the transformation of ASN.1
combining type definitions into a grammar.

25.1.2. Unique Component Attribution

This section describes conditions that the grammar must satisfy so
that each element and attribute in a received RXER encoding can be
uniquely associated with an ASN.1 component definition.

Definition (used by the grammar): A non-terminal, N, is used by the
grammar if:

(1) N is the start symbol or

(2) N appears on the right-hand side of a production where the
    non-terminal on the left-hand side is used by the grammar.

   Definition (multiple derivation paths): A non-terminal, N, has
   multiple derivation paths if:

   (1) N appears on the right-hand side of a production where the
       non-terminal on the left-hand side has multiple derivation paths,
       or

   (2) N appears on the right-hand side of more than one production
       where the non-terminal on the left-hand side is used by the
       grammar, or

   (3) N is the start symbol and it appears on the right-hand side of a
       production where the non-terminal on the left-hand side is used
       by the grammar.

   For every ASN.1 type with a base type containing components that are
   subject to a GROUP encoding instruction, the grammar derived by the
   method described in this document MUST NOT have:

   (1) two or more primary non-terminals that are used by the grammar
       and are associated with element components having the same
       expanded name, or

   (2) two or more primary non-terminals that are used by the grammar
       and are associated with attribute components having the same
       expanded name, or

   (3) a primary non-terminal that has multiple derivation paths and is
       associated with an attribute component.

      Aside: Case (1) is in response to component referencing notations
      that are evaluated with respect to the XML encoding of an abstract
      value. Case (1) guarantees, without having to do extensive
      testing (which would necessarily have to take account of encoding
      instructions for all other encoding rules), that all sibling
      elements with the same expanded name will be associated with
      equivalent type definitions. Such equivalence allows a component
      referenced by element name to be re-encoded using a different set
      of ASN.1 encoding rules without ambiguity as to which type
      definition and encoding instructions apply.

      Cases (2) and (3) ensure that an attribute name is always uniquely
      associated with one component that can occur at most once and is
      always nested in the same part of an abstract value.

   Example

      The following example types illustrate various uses and misuses of
      the GROUP encoding instruction with respect to unique component
      attribution:

         TA ::= SEQUENCE {
             a [GROUP] TB,
             b [GROUP] CHOICE {
                 a [GROUP] TB,
                 b [NAME AS "c"] [ATTRIBUTE] INTEGER,
                 c INTEGER,
                 d TB,
                 e [GROUP] TD,
                 f [ATTRIBUTE] UTF8String
             },
             c [ATTRIBUTE] INTEGER,
             d [GROUP] SEQUENCE OF
                 a [GROUP] SEQUENCE {
                     a [ATTRIBUTE] OBJECT IDENTIFIER,
                     b INTEGER
                 },
             e [NAME AS "c"] INTEGER,
             COMPONENTS OF TD
         }

         TB ::= SEQUENCE {
             a INTEGER,
             b [ATTRIBUTE] BOOLEAN,
             COMPONENTS OF TC
         }

         TC ::= SEQUENCE {
             f OBJECT IDENTIFIER
         }

         TD ::= SEQUENCE {
             g OBJECT IDENTIFIER
         }

      The grammar for TA is constructed after performing the
      COMPONENTS OF transformation. The result of this transformation
      is shown next. This example will depart from the usual convention
      of using just the identifier of a NamedType to represent the
      primary non-terminal for that NamedType. A label relative to the
      outermost type will be used instead to better illustrate unique
      component attribution. The labels used for the non-terminals are
      shown down the right-hand side.

```
        TA ::= SEQUENCE {
            a  [GROUP] TB,                              -- TA.a
            b  [GROUP] CHOICE {                         -- TA.b
                a  [GROUP] TB,                          -- TA.b.a
                b  [NAME AS "c"] [ATTRIBUTE] INTEGER,   -- TA.b.b
                c  INTEGER,                             -- TA.b.c
                d  TB,                                  -- TA.b.d
                e  [GROUP] TD,                          -- TA.b.e
                f  [ATTRIBUTE] UTF8String               -- TA.b.f
            },
            c  [ATTRIBUTE] INTEGER,                     -- TA.c
            d  [GROUP] SEQUENCE OF                      -- TA.d
               a [GROUP] SEQUENCE {                     -- TA.d.a
                   a  [ATTRIBUTE] OBJECT IDENTIFIER,    -- TA.d.a.a
                   b  INTEGER                           -- TA.d.a.b
               },
            e  [NAME AS "c"] INTEGER,                   -- TA.e
            g  OBJECT IDENTIFIER                        -- TA.g
        }

        TB ::= SEQUENCE {
            a  INTEGER,                                 -- TB.a
            b  [ATTRIBUTE] BOOLEAN,                     -- TB.b
            f  OBJECT IDENTIFIER                        -- TB.f
        }

        -- Type TC is no longer of interest. --

        TD ::= SEQUENCE {
            g  OBJECT IDENTIFIER                        -- TD.g
        }
```

   The associated grammar is:

```
     S ::= TA.a TA.b TA.c TA.d TA.e TA.g

     TA.a ::= TB.a TB.b TB.f

     TB.a ::= "a"
     TB.b ::= "@b"
     TB.f ::= "f"

     TA.b ::= TA.b.a
     TA.b ::= TA.b.b
     TA.b ::= TA.b.c
     TA.b ::= TA.b.d
     TA.b ::= TA.b.e
     TA.b ::= TA.b.f
```

```
TA.b.a ::= TB.a TB.b TB.f
TA.b.b ::= "@c"
TA.b.c ::= "c"
TA.b.d ::= "d"
TA.b.e ::= TD.g
TA.b.f ::= "@f"

TD.g ::= "g"

TA.c ::= "@c"

TA.d ::= TA.d.a TA.d
TA.d ::=

TA.d.a ::= TA.d.a.a TA.d.a.b

TA.d.a.a := "@a"
TA.d.a.b ::= "b"

TA.e ::= "c"

TA.g ::= "g"
```

All the non-terminals are used by the grammar.

The type definition for TA is invalid because there are two
instances where two or more primary non-terminals are associated
with element components having the same expanded name:

(1) TA.b.c and TA.e (both generate the terminal "c"), and

(2) TD.g and TA.g (both generate the terminal "g").

In case (2), TD.g and TA.g are derived from the same instance of
NamedType notation, but become distinct components following the
COMPONENTS OF transformation. AUTOMATIC tagging is applied after
the COMPONENTS OF transformation, which means that the types of
the components corresponding to TD.g and TA.g will end up with
different tags, and therefore the types will not be equivalent.

The type definition for TA is also invalid because there is one
instance where two or more primary non-terminals are associated
with attribute components having the same expanded name: TA.b.b
and TA.c (both generate the terminal "@c").

The non-terminals with multiple derivation paths are: TA.d,
TA.d.a, TA.d.a.a, TA.d.a.b, TB.a, TB.b, and TB.f. The type
definition for TA is also invalid because TA.d.a.a and TB.b are
primary non-terminals that are associated with an attribute
component.

25.1.3. Deterministic Grammars

Let the First Set of a production P, denoted First(P), be the set of
all element terminals T where T is the first element terminal in a
sequence of terminals that can be generated from the right-hand side
of P. There can be any number of leading attribute terminals before
T.

Let the Follow Set of a non-terminal N, denoted Follow(N), be the set
of all element terminals T where T is the first element terminal
following N in a sequence of non-terminals and terminals that can be
generated from the grammar. There can be any number of attribute
terminals between N and T. If a sequence of non-terminals and
terminals can be generated from the grammar where N is not followed
by any element terminals, then Follow(N) also contains a special end
terminal, denoted by "$".

   Aside: If N does not appear on the right-hand side of any
   production, then Follow(N) will be empty.

For a production P, let the predicate Empty(P) be true if and only if
the empty sequence of terminals can be generated from P. Otherwise,
Empty(P) is false.

Definition (base grammar): The base grammar is a rewriting of the
grammar in which the non-terminals for every ExtensionAddition and
ExtensionAdditionAlternative are removed from the right-hand side of
all productions.

For a production P, let the predicate Preselected(P) be true if and
only if every sequence of terminals that can be generated from the
right-hand side of P using only the base grammar contains at least
one attribute terminal. Otherwise, Preselected(P) is false.

The Select Set of a production P, denoted Select(P), is empty if
Preselected(P) is true; otherwise, it contains First(P). Let N be
the non-terminal on the left-hand side of P. If Empty(P) is true,
then Select(P) also contains Follow(N).

Aside: It may appear somewhat dubious to include the attribute
components in the grammar because, in reality, attributes appear
unordered within the start tag of an element, and not interspersed
with the child elements as the grammar would suggest. This is why
attribute terminals are ignored in composing the First Sets and
Follow Sets. However, the attribute terminals are important in
composing the Select Sets because they can preselect a production
and can prevent a production from being able to generate an empty
sequence of terminals. In real terms, this corresponds to an RXER
decoder using the attributes to determine the presence or absence
of optional components and to select between the alternatives of a
CHOICE, even before considering the child elements.

An attribute appearing in an extension isn't used to preselect a
production since, in general, a decoder using an earlier version
of the specification would not be able to associate the attribute
with any particular extension insertion point.

Let the Reach Set of a non-terminal N, denoted Reach(N), be the set
of all element terminals T where T appears in a sequence of terminals
that can be generated from N.

Aside: It can be readily shown that all the optional attribute
components and all but one of the mandatory attribute components
of a SEQUENCE or SET type can be ignored in constructing the
grammar because their omission does not alter the First, Follow,
Select, or Reach Sets, or the evaluation of the Preselected and
Empty predicates.

A grammar is deterministic (for the purposes of an RXER decoder) if
and only if:

(1) there do not exist two productions P and Q, with the same
    non-terminal on the left-hand side, where the intersection of
    Select(P) and Select(Q) is not empty, and

(2) there does not exist a non-terminal E for an ExtensionAddition or
    ExtensionAdditionAlternative where the intersection of Reach(E)
    and Follow(E) is not empty.

Aside: In case (1), if the intersection is not empty, then a
decoder would have two or more possible ways to attempt to decode
the input into an abstract value. In case (2), if the
intersection is not empty, then a decoder using an earlier version
of the ASN.1 specification would confuse an element in an unknown
(to that decoder) extension with a known component following the
extension.

Aside: In the absence of any attribute components, case (1) is the test for an LL(1) grammar.

For every ASN.1 type with a base type containing components that are subject to a GROUP encoding instruction, the grammar derived by the method described in this document MUST be deterministic.

25.1.4. Attributes in Unknown Extensions

An insertion point production is able to accept unknown attributes if the non-terminal on the left-hand side of the production does not have multiple derivation paths.

Aside: If the non-terminal has multiple derivation paths, then any future extension cannot possibly contain an attribute component because that would violate the requirements of Section 25.1.2.

For a deterministic grammar, there is only one possible way to construct a sequence of element terminals matching the element content of an element in a correctly formed RXER encoding. Any unknown attributes of the element are accepted if at least one insertion point production that is able to accept unknown attributes is used in that construction.

Example

Consider this type definition:

```
CHOICE {
    one  UTF8String,
    two  [GROUP] SEQUENCE {
        three  INTEGER,
        ...
    }
}
```

The associated grammar is:

```
S ::= one
S ::= two

two ::= three I1

I1 ::= "*" I1
I1 ::=

one ::= "one"
three ::= "three"
```

The third production is an insertion point production, and it is able to accept unknown attributes.

When decoding a value of this type, if the element content contains a <one> child element, then any unrecognized attribute would be illegal as the insertion point production would not be used to recognize the input (the "one" alternative does not admit an extension insertion point). If the element content contains a <three> element, then an unrecognized attribute would be accepted because the insertion point production would be used to recognize the input (the "two" alternative that generates the <three> element has an extensible type).

If the SEQUENCE type were prefixed by a NO-INSERTIONS encoding instruction, then the third, fourth, and fifth productions would be replaced by:

    two ::= three

With this change, any unrecognized attribute would be illegal for the "two" alternative also, since the replacement production is not an insertion point production.

If more than one insertion point production that is able to accept unknown attributes is used in constructing a matching sequence of element terminals, then a decoder is free to associate an unrecognized attribute with any one of the extension insertion points corresponding to those insertion point productions. The justification for doing so comes from the following two observations:

(1) If the encoding of an abstract value contains an extension where the type of the extension is unknown to the receiver, then it is generally impossible to re-encode the value using a different set of encoding rules, including the canonical variant of the received encoding. This is true no matter which encoding rules are being used. It is desirable for a decoder to be able to accept and store the raw encoding of an extension without raising an error, and to re-insert the raw encoding of the extension when re-encoding the abstract value using the same non-canonical encoding rules. However, there is little more that an application can do with an unknown extension.

    An application using RXER can successfully accept, store, and re-encode an unrecognized attribute regardless of which extension insertion point it might be ascribed to.

   (2) Even if there is a single extension insertion point, an unknown
       extension could still be the encoding of a value of any one of an
       infinite number of valid type definitions. For example, an
       attribute or element component could be nested to any arbitrary
       depth within CHOICEs whose components are subject to GROUP
       encoding instructions.

          Aside: A similar series of nested CHOICEs could describe an
          unknown extension in a Basic Encoding Rules (BER) encoding
          [X.690].

26. Security Considerations

   ASN.1 compiler implementors should take special care to be thorough
   in checking that the GROUP encoding instruction has been correctly
   used; otherwise, ASN.1 specifications with ambiguous RXER encodings
   could be deployed.

   Ambiguous encodings mean that the abstract value recovered by a
   decoder may differ from the original abstract value that was encoded.
   If that is the case, then a digital signature generated with respect
   to the original abstract value (using a canonical encoding other than
   CRXER) will not be successfully verified by a receiver using the
   decoded abstract value. Also, an abstract value may have
   security-sensitive fields, and in particular, fields used to grant or
   deny access. If the decoded abstract value differs from the encoded
   abstract value, then a receiver using the decoded abstract value will
   be applying different security policy than that embodied in the
   original abstract value.

27. References

27.1. Normative References

   [BCP14]   Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [URI]   Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform
              Resource Identifiers (URI): Generic Syntax", STD 66, RFC
              3986, January 2005.

   [RXER]   Legg, S. and D. Prager, "Robust XML Encoding Rules (RXER)
              for Abstract Syntax Notation One (ASN.1)", RFC 4910, July
              2007.

   [ASN.X]   Legg, S., "Abstract Syntax Notation X (ASN.X)", RFC 4912,
              July 2007.

   [X.680]   ITU-T Recommendation X.680 (07/02) | ISO/IEC 8824-1,
               Information technology - Abstract Syntax Notation One
               (ASN.1): Specification of basic notation.

   [X.680-1]   ITU-T Recommendation X.680 (2002) Amendment 1 (10/03) |
               ISO/IEC 8824-1:2002/Amd 1:2004, Support for EXTENDED-XER.

   [X.683]   ITU-T Recommendation X.683 (07/02) | ISO/IEC 8824-4,
               Information technology - Abstract Syntax Notation One
               (ASN.1): Parameterization of ASN.1 specifications.

   [XML10]   Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E. and
               F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fourth
               Edition)", W3C Recommendation,
               http://www.w3.org/TR/2006/REC-xml-20060816, August 2006.

   [XMLNS10]   Bray, T., Hollander, D., Layman, A., and R. Tobin,
               "Namespaces in XML 1.0 (Second Edition)", W3C
               Recommendation,
               http://www.w3.org/TR/2006/REC-xml-names-20060816, August
               2006.

   [XSD1]   Thompson, H., Beech, D., Maloney, M. and N. Mendelsohn,
               "XML Schema Part 1: Structures Second Edition", W3C
               Recommendation,
               http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/,
               October 2004.

   [XSD2]   Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes
               Second Edition", W3C Recommendation,
               http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/,
               October 2004.

   [RNG]   Clark, J. and M. Makoto, "RELAX NG Tutorial", OASIS
               Committee Specification, http://www.oasis-open.org/
               committees/relax-ng/tutorial-20011203.html, December 2001.

27.2. Informative References

   [INFOSET]   Cowan, J. and R. Tobin, "XML Information Set (Second
               Edition)", W3C Recommendation, http://www.w3.org/
               TR/2004/REC-xml-infoset-20040204, February 2004.

   [X.690]   ITU-T Recommendation X.690 (07/02) | ISO/IEC 8825-1,
               Information technology - ASN.1 encoding rules:
               Specification of Basic Encoding Rules (BER), Canonical
               Encoding Rules (CER) and Distinguished Encoding Rules
               (DER).

Appendix A. GROUP Encoding Instruction Examples

   This appendix is non-normative.

   This appendix contains examples of both correct and incorrect use of
   the GROUP encoding instruction, determined with respect to the
   grammars derived from the example type definitions. The productions
   of the grammars are labeled for convenience. Sets and predicates for
   non-terminals with only one production will be omitted from the
   examples since they never indicate non-determinism.

   The requirements of Section 25.1.2 ("Unique Component Attribution")
   are satisfied by all the examples in this appendix and the appendices
   that follow it.

A.1. Example 1

   Consider this type definition:

```
        SEQUENCE {
            one     [GROUP] SEQUENCE {
                two     UTF8String OPTIONAL
            } OPTIONAL,
            three  INTEGER
        }
```

   The associated grammar is:

```
        P1:  S ::= one three
        P2:  one ::= two
        P3:  one ::=
        P4:  two ::= "two"
        P5:  two ::=
        P6:  three ::= "three"
```

   Select Sets have to be evaluated to test the validity of the type
   definition. The grammar leads to the following sets and predicates:

```
        First(P2) = { "two" }
        First(P3) = { }
        Preselected(P2) = Preselected(P3) = false
        Empty(P2) = Empty(P3) = true
        Follow(one) = { "three" }
        Select(P2) = First(P2) + Follow(one) = { "two", "three" }
        Select(P3) = First(P3) + Follow(one) = { "three" }
```

```
    First(P4) = { "two" }
    First(P5) = { }
    Preselected(P4) = Preselected(P5) = Empty(P4) = false
    Empty(P5) = true
    Follow(two) = { "three" }
    Select(P4) = First(P4) = { "two" }
    Select(P5) = First(P5) + Follow(two) = { "three" }
```

The intersection of Select(P2) and Select(P3) is not empty; hence, the grammar is not deterministic, and the type definition is not valid. If the RXER encoding of a value of the type does not have a child element <two>, then it is not possible to determine whether the "one" component is present or absent in the value.

Now consider this type definition with attributes in the "one" component:

```
    SEQUENCE {
        one    [GROUP] SEQUENCE {
            two    UTF8String OPTIONAL,
            four   [ATTRIBUTE] BOOLEAN,
            five   [ATTRIBUTE] BOOLEAN OPTIONAL
        } OPTIONAL,
        three  INTEGER
    }
```

The associated grammar is:

```
    P1:  S ::= one three
    P2:  one ::= two four five
    P3:  one ::=
    P4:  two ::= "two"
    P5:  two ::=
    P6:  four ::= "@four"
    P7:  five ::= "@five"
    P8:  five ::=
    P9:  three ::= "three"
```

This grammar leads to the following sets and predicates:

```
    First(P2) = { "two" }
    First(P3) = { }
    Preselected(P3) = Empty(P2) = false
    Preselected(P2) = Empty(P3) = true
    Follow(one) = { "three" }
    Select(P2) = { }
    Select(P3) = First(P3) + Follow(one) = { "three" }
```

```
     First(P4) = { "two" }
     First(P5) = { }
     Preselected(P4) = Preselected(P5) = Empty(P4) = false
     Empty(P5) = true
     Follow(two) = { "three" }
     Select(P4) = First(P4) = { "two" }
     Select(P5) = First(P5) + Follow(two) = { "three" }

     First(P7) = { }
     First(P8) = { }
     Preselected(P8) = Empty(P7) = false
     Preselected(P7) = Empty(P8) = true
     Follow(five) = { "three" }
     Select(P7) = { }
     Select(P8) = First(P8) + Follow(five) = { "three" }
```

   The intersection of Select(P2) and Select(P3) is empty, as is the
   intersection of Select(P4) and Select(P5) and the intersection of
   Select(P7) and Select(P8); hence, the grammar is deterministic, and
   the type definition is valid. In a correct RXER encoding, the "one"
   component will be present if and only if the "four" attribute is
   present.

A.2. Example 2

   Consider this type definition:

```
     CHOICE {
         one     [GROUP] SEQUENCE {
             two     [ATTRIBUTE] BOOLEAN OPTIONAL
         },
         three  INTEGER,
         four    [GROUP] SEQUENCE {
             five    BOOLEAN OPTIONAL
         }
     }
```

   The associated grammar is:

```
     P1:  S ::= one
     P2:  S ::= three
     P3:  S ::= four
     P4:  one ::= two
     P5:  two ::= "@two"
     P6:  two ::=
     P7:  three ::= "three"
     P8:  four ::= five
     P9:  five ::= "five"
```

```
     P10: five ::=
```

   This grammar leads to the following sets and predicates:

```
     First(P1) = { }
     First(P2) = { "three" }
     First(P3) = { "five" }
     Preselected(P1) = Preselected(P2) = Preselected(P3) = false
     Empty(P2) = false
     Empty(P1) = Empty(P3) = true
     Follow(S) = { "$" }
     Select(P1) = First(P1) + Follow(S) = { "$" }
     Select(P2) = First(P2) = { "three" }
     Select(P3) = First(P3) + Follow(S) = { "five", "$" }

     First(P5) = { }
     First(P6) = { }
     Preselected(P6) = Empty(P5) = false
     Preselected(P5) = Empty(P6) = true
     Follow(two) = { "$" }
     Select(P5) = { }
     Select(P6) = First(P6) + Follow(two) = { "$" }

     First(P9) = { "five" }
     First(P10) = { }
     Preselected(P9) = Preselected(P10) = Empty(P9) = false
     Empty(P10) = true
     Follow(five) = { "$" }
     Select(P9) = First(P9) = { "five" }
     Select(P10) = First(P10) + Follow(five) = { "$" }
```

   The intersection of Select(P1) and Select(P3) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid. If the RXER encoding of a value of the type is empty, then it
   is not possible to determine whether the "one" alternative or the
   "four" alternative has been chosen.

   Now consider this slightly different type definition:

```
     CHOICE {
         one     [GROUP] SEQUENCE {
             two     [ATTRIBUTE] BOOLEAN
         },
         three  INTEGER,
         four    [GROUP] SEQUENCE {
             five    BOOLEAN OPTIONAL
         }
     }
```

The associated grammar is:

```
P1:  S ::= one
P2:  S ::= three
P3:  S ::= four
P4:  one ::= two
P5:  two ::= "@two"
P6:  three ::= "three"
P7:  four ::= five
P8:  five ::= "five"
P9:  five ::=
```

This grammar leads to the following sets and predicates:

```
First(P1) = { }
First(P2) = { "three" }
First(P3) = { "five" }
Preselected(P2) = Preselected(P3) = false
Empty(P1) = Empty(P2) = false
Preselected(P1) = Empty(P3) = true
Follow(S) = { "$" }
Select(P1) = { }
Select(P2) = First(P2) = { "three" }
Select(P3) = First(P3) + Follow(S) = { "five", "$" }

First(P8) = { "five" }
First(P9) = { }
Preselected(P8) = Preselected(P9) = Empty(P8) = false
Empty(P9) = true
Follow(five) = { "$" }
Select(P8) = First(P8) = { "five" }
Select(P9) = First(P9) + Follow(five) = { "$" }
```

The intersection of Select(P1) and Select(P2) is empty, the
intersection of Select(P1) and Select(P3) is empty, the intersection
of Select(P2) and Select(P3) is empty, and the intersection of
Select(P8) and Select(P9) is empty; hence, the grammar is
deterministic, and the type definition is valid. The "one" and
"four" alternatives can be distinguished because the "one"
alternative has a mandatory attribute.

A.3. Example 3

   Consider this type definition:

```
SEQUENCE {
    one  [GROUP] CHOICE {
        two    [ATTRIBUTE] BOOLEAN,
        three  [GROUP] SEQUENCE OF number INTEGER
    } OPTIONAL
}
```

   The associated grammar is:

```
P1:  S ::= one
P2:  one ::= two
P3:  one ::= three
P4:  one ::=
P5:  two ::= "@two"
P6:  three ::= number three
P7:  three ::=
P8:  number ::= "number"
```

   This grammar leads to the following sets and predicates:

```
First(P2) = { }
First(P3) = { "number" }
First(P4) = { }
Preselected(P3) = Preselected(P4) = Empty(P2) = false
Preselected(P2) = Empty(P3) = Empty(P4) = true
Follow(one) = { "$" }
Select(P2) = { }
Select(P3) = First(P3) + Follow(one) = { "number", "$" }
Select(P4) = First(P4) + Follow(one) = { "$" }

First(P6) = { "number" }
First(P7) = { }
Preselected(P6) = Preselected(P7) = Empty(P6) = false
Empty(P7) = true
Follow(three) = { "$" }
Select(P6) = First(P6) = { "number" }
Select(P7) = First(P7) + Follow(three) = { "$" }
```

   The intersection of Select(P3) and Select(P4) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid. If the RXER encoding of a value of the type is empty, then it
   is not possible to determine whether the "one" component is absent or
   the empty "three" alternative has been chosen.

A.4. Example 4

   Consider this type definition:

       SEQUENCE {
           one [GROUP] CHOICE {
               two   [ATTRIBUTE] BOOLEAN,
               three [ATTRIBUTE] BOOLEAN
           } OPTIONAL
       }

   The associated grammar is:

       P1: S ::= one
       P2: one ::= two
       P3: one ::= three
       P4: one ::=
       P5: two ::= "@two"
       P6: three ::= "@three"

   This grammar leads to the following sets and predicates:

       First(P2) = { }
       First(P3) = { }
       First(P4) = { }
       Preselected(P4) = Empty(P2) = Empty(P3) = false
       Preselected(P2) = Preselected(P3) = Empty(P4) = true
       Follow(one) = { "$" }
       Select(P2) = { }
       Select(P3) = { }
       Select(P4) = First(P4) + Follow(one) = { "$" }

   The intersection of Select(P2) and Select(P3) is empty, the
   intersection of Select(P2) and Select(P4) is empty, and the
   intersection of Select(P3) and Select(P4) is empty; hence, the
   grammar is deterministic, and the type definition is valid.

A.5. Example 5

   Consider this type definition:

       SEQUENCE {
           one [GROUP] SEQUENCE OF number INTEGER OPTIONAL
       }

The associated grammar is:

```
P1:  S ::= one
P2:  one ::= number one
P3:  one ::=
P4:  one ::=
P5:  number ::= "number"
```

P3 is generated during the processing of the SEQUENCE OF type. P4 is
generated because the "one" component is optional.

This grammar leads to the following sets and predicates:

```
First(P2) = { "number" }
First(P3) = { }
First(P4) = { }
Preselected(P2) = Preselected(P3) = Preselected(P4) = false
Empty(P2) = false
Empty(P3) = Empty(P4) = true
Follow(one) = { "$" }
Select(P2) = First(P2) = { "number" }
Select(P3) = First(P3) + Follow(one) = { "$" }
Select(P4) = First(P4) + Follow(one) = { "$" }
```

The intersection of Select(P3) and Select(P4) is not empty; hence,
the grammar is not deterministic, and the type definition is not
valid. If the RXER encoding of a value of the type does not have any
<number> child elements, then it is not possible to determine whether
the "one" component is present or absent in the value.

Consider this similar type definition with a SIZE constraint:

```
SEQUENCE {
    one  [GROUP] SEQUENCE SIZE(1..MAX) OF number INTEGER OPTIONAL
}
```

The associated grammar is:

```
P1:  S ::= one
P2:  one ::= number one'
P3:  one' ::= number one'
P4:  one' ::=
P5:  one ::=
P6:  number ::= "number"
```

This grammar leads to the following sets and predicates:

```
First(P2) = { "number" }
First(P5) = { }
Preselected(P2) = Preselected(P5) = Empty(P2) = false
Empty(P5) = true
Follow(one) = { "$" }
Select(P2) = First(P2) = { "number" }
Select(P5) = First(P5) + Follow(one) = { "$" }

First(P3) = { "number" }
First(P4) = { }
Preselected(P3) = Preselected(P4) = Empty(P3) = false
Empty(P4) = true
Follow(one') = { "$" }
Select(P3) = First(P3) = { "number" }
Select(P4) = First(P4) + Follow(one') = { "$" }
```

The intersection of Select(P2) and Select(P5) is empty, as is the
intersection of Select(P3) and Select(P4); hence, the grammar is
deterministic, and the type definition is valid. If there are no
<number> child elements, then the "one" component is necessarily
absent and there is no ambiguity.

A.6. Example 6

Consider this type definition:

```
SEQUENCE {
    beginning  [GROUP] List,
    middle     UTF8String OPTIONAL,
    end        [GROUP] List
}

List ::= SEQUENCE OF string UTF8String
```

The associated grammar is:

```
P1:  S ::= beginning middle end
P2:  beginning ::= string beginning
P3:  beginning ::=
P4:  middle ::= "middle"
P5:  middle ::=
P6:  end ::= string end
P7:  end ::=
P8:  string ::= "string"
```

   This grammar leads to the following sets and predicates:

```
First(P2) = { "string" }
First(P3) = { }
Preselected(P2) = Preselected(P3) = Empty(P2) = false
Empty(P3) = true
Follow(beginning) = { "middle", "string", "$" }
Select(P2) = First(P2) = { "string" }
Select(P3) = First(P3) + Follow(beginning)
           = { "middle", "string", "$" }

First(P4) = { "middle" }
First(P5) = { }
Preselected(P4) = Preselected(P5) = Empty(P4) = false
Empty(P5) = true
Follow(middle) = { "string", "$" }
Select(P4) = First(P4) = { "middle" }
Select(P5) = First(P5) + Follow(middle) = { "string", "$" }

First(P6) = { "string" }
First(P7) = { }
Preselected(P6) = Preselected(P7) = Empty(P6) = false
Empty(P7) = true
Follow(end) = { "$" }
Select(P6) = First(P6) = { "string" }
Select(P7) = First(P7) + Follow(end) = { "$" }
```

   The intersection of Select(P2) and Select(P3) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid.

   Now consider the following type definition:

```
SEQUENCE {
    beginning      [GROUP] List,
    middleAndEnd   [GROUP] SEQUENCE {
        middle         UTF8String,
        end            [GROUP] List
    } OPTIONAL
}
```

   The associated grammar is:

```
P1:  S ::= beginning middleAndEnd
P2:  beginning ::= string beginning
P3:  beginning ::=
P4:  middleAndEnd ::= middle end
P5:  middleAndEnd ::=
```

```
     P6:  middle ::= "middle"
     P7:  end ::= string end
     P8:  end ::=
     P9:  string ::= "string"
```

   This grammar leads to the following sets and predicates:

```
     First(P2) = { "string" }
     First(P3) = { }
     Preselected(P2) = Preselected(P3) = Empty(P2) = false
     Empty(P3) = true
     Follow(beginning) = { "middle", "$" }
     Select(P2) = First(P2) = { "string" }
     Select(P3) = First(P3) + Follow(beginning) = { "middle", "$" }

     First(P4) = { "middle" }
     First(P5) = { }
     Preselected(P4) = Preselected(P5) = Empty(P4) = false
     Empty(P5) = true
     Follow(middleAndEnd) = { "$" }
     Select(P4) = First(P4) = { "middle" }
     Select(P5) = First(P5) + Follow(middleAndEnd) = { "$" }

     First(P7) = { "string" }
     First(P8) = { }
     Preselected(P7) = Preselected(P8) = Empty(P7) = false
     Empty(P8) = true
     Follow(end) = { "$" }
     Select(P7) = First(P7) = { "string" }
     Select(P8) = First(P8) + Follow(end) = { "$" }
```

   The intersection of Select(P2) and Select(P3) is empty, as is the
   intersection of Select(P4) and Select(P5) and the intersection of
   Select(P7) and Select(P8); hence, the grammar is deterministic, and
   the type definition is valid.

A.7. Example 7

   Consider the following type definition:

```
     SEQUENCE SIZE(1..MAX) OF
         one  [GROUP] SEQUENCE {
             two    INTEGER OPTIONAL
         }
```

   The associated grammar is:

```
    P1:  S ::= one S'
    P2:  S' ::= one S'
    P3:  S' ::=
    P4:  one ::= two
    P5:  two ::= "two"
    P6:  two ::=
```

   This grammar leads to the following sets and predicates:

```
    First(P2) = { "two" }
    First(P3) = { }
    Preselected(P2) = Preselected(P3) = false
    Empty(P2) = Empty(P3) = true
    Follow(S') = { "$" }
    Select(P2) = First(P2) + Follow(S') = { "two", "$" }
    Select(P3) = First(P3) + Follow(S') = { "$" }

    First(P5) = { "two" }
    First(P6) = { }
    Preselected(P5) = Preselected(P6) = Empty(P5) = false
    Empty(P6) = true
    Follow(two) = { "two", "$" }
    Select(P5) = First(P5) = { "two" }
    Select(P6) = First(P6) + Follow(two) = { "two", "$" }
```

   The intersection of Select(P2) and Select(P3) is not empty and the
   intersection of Select(P5) and Select(P6) is not empty; hence, the
   grammar is not deterministic, and the type definition is not valid.
   The encoding of a value of the type contains an indeterminate number
   of empty instances of the component type.

A.8.   Example 8

   Consider the following type definition:

```
    SEQUENCE OF
        list [GROUP] SEQUENCE SIZE(1..MAX) OF number INTEGER
```

   The associated grammar is:

```
    P1:  S ::= list S
    P2:  S ::=
    P3:  list ::= number list'
    P4:  list' ::= number list'
    P5:  list' ::=
    P6:  number ::= "number"
```

   This grammar leads to the following sets and predicates:

```
    First(P1) = { "number" }
    First(P2) = { }
    Preselected(P1) = Preselected(P2) = Empty(P1) = false
    Empty(P2) = true
    Follow(S) = { "$" }
    Select(P1) = First(P1) = { "number" }
    Select(P2) = First(P2) + Follow(S) = { "$" }

    First(P4) = { "number" }
    First(P5) = { }
    Preselected(P4) = Preselected(P5) = Empty(P4) = false
    Empty(P5) = true
    Follow(list') = { "number", "$" }
    Select(P4) = First(P4) = { "number" }
    Select(P5) = First(P5) + Follow(list') = { "number", "$" }
```

   The intersection of Select(P4) and Select(P5) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid. The type describes a list of lists, but it is not possible
   for a decoder to determine where the outer lists begin and end.

A.9. Example 9

   Consider the following type definition:

```
    SEQUENCE OF item [GROUP] SEQUENCE {
        before  [GROUP] OneAndTwo,
        core    UTF8String,
        after   [GROUP] OneAndTwo OPTIONAL
    }

    OneAndTwo ::= SEQUENCE {
        non-core  UTF8String
    }
```

   The associated grammar is:

```
    P1:  S ::= item S
    P2:  S ::=
    P3:  item ::= before core after
    P4:  before ::= non-core
    P5:  non-core ::= "non-core"
    P6:  core ::= "core"
    P7:  after ::= non-core
    P8:  after ::=
```

   This grammar leads to the following sets and predicates:

```
    First(P1) = { "non-core" }
    First(P2) = { }
    Preselected(P1) = Preselected(P2) = Empty(P1) = false
    Empty(P2) = true
    Follow(S) = { "$" }
    Select(P1) = First(P1) = { "non-core" }
    Select(P2) = First(P2) + Follow(S) = { "$" }

    First(P7) = { "non-core" }
    First(P8) = { }
    Preselected(P7) = Preselected(P8) = Empty(P7) = false
    Empty(P8) = true
    Follow(after) = { "non-core", "$" }
    Select(P7) = First(P7) = { "non-core" }
    Select(P8) = First(P8) + Follow(after) = { "non-core", "$" }
```

   The intersection of Select(P7) and Select(P8) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid. There is ambiguity between the end of one item and the start
   of the next. Without looking ahead in an encoding, it is not
   possible to determine whether a <non-core> element belongs with the
   preceding or following <core> element.

A.10. Example 10

   Consider the following type definition:

```
    CHOICE {
        one    [GROUP] List,
        two    [GROUP] SEQUENCE {
            three  [ATTRIBUTE] UTF8String,
            four   [GROUP] List
        }
    }

    List ::= SEQUENCE OF string UTF8String
```

   The associated grammar is:

```
    P1:  S ::= one
    P2:  S ::= two
    P3:  one ::= string one
    P4:  one ::=
    P5:  two ::= three four
    P6:  three ::= "@three"
    P7:  four ::= string four
```

```
P8:  four ::=
P9:  string ::= "string"
```

This grammar leads to the following sets and predicates:

```
First(P1) = { "string" }
First(P2) = { "string" }
Preselected(P1) = Empty(P2) = false
Preselected(P2) = Empty(P1) = true
Follow(S) = { "$" }
Select(P1) = First(P1) + Follow(S) = { "string", "$" }
Select(P2) = { }

First(P3) = { "string" }
First(P4) = { }
Preselected(P3) = Preselected(P4) = Empty(P3) = false
Empty(P4) = true
Follow(one) = { "$" }
Select(P3) = First(P3) = { "string" }
Select(P4) = First(P4) + Follow(one) = { "$" }

First(P7) = { "string" }
First(P8) = { }
Preselected(P7) = Preselected(P8) = Empty(P7) = false
Empty(P8) = true
Follow(four) = { "$" }
Select(P7) = First(P7) = { "string" }
Select(P8) = First(P8) + Follow(four) = { "$" }
```

The intersection of Select(P1) and Select(P2) is empty, as is the
intersection of Select(P3) and Select(P4) and the intersection of
Select(P7) and Select(P8); hence, the grammar is deterministic, and
the type definition is valid. Although both alternatives of the
CHOICE can begin with a <string> element, an RXER decoder would use
the presence of a "three" attribute to decide whether to select or
disregard the "two" alternative.

However, an attribute in an extension cannot be used to select
between alternatives. Consider the following type definition:

```
    [SINGULAR-INSERTIONS] CHOICE {
        one   [GROUP] List,
        ...,
        two   [GROUP] SEQUENCE {
            three  [ATTRIBUTE] UTF8String,
            four   [GROUP] List
        } -- ExtensionAdditionAlternative (E1).
        -- The extension insertion point is here (I1).
    }

    List ::= SEQUENCE OF string UTF8String
```

The associated grammar is:

```
    P1:  S ::= one
    P10: S ::= E1
    P11: S ::= "*"
    P12: E1 ::= two
    P3:  one ::= string one
    P4:  one ::=
    P5:  two ::= three four
    P6:  three ::= "@three"
    P7:  four ::= string four
    P8:  four ::=
    P9:  string ::= "string"
```

This grammar leads to the following sets and predicates for P1, P10
and P11:

```
    First(P1) = { "string" }
    First(P10) = { "string" }
    First(P11) = { "*" }
    Preselected(P1) = Preselected(P10) = Preselected(P11) = false
    Empty(P10) = Empty(P11) = false
    Empty(P1) = true
    Follow(S) = { "$" }
    Select(P1) = First(P1) + Follow(S) = { "string", "$" }
    Select(P10) = First(P10) = { "string" }
    Select(P11) = First(P11) = { "*" }
```

Preselected(P10) evaluates to false because Preselected(P10) is
evaluated on the base grammar, wherein P10 is rewritten as:

```
    P10: S ::=
```

The intersection of Select(P1) and Select(P10) is not empty; hence, the grammar is not deterministic, and the type definition is not valid. An RXER decoder using the original, unextended version of the definition would not know that the "three" attribute selects between the "one" alternative and the extension.

Appendix B. Insertion Encoding Instruction Examples

This appendix is non-normative.

This appendix contains examples showing the use of insertion encoding instructions to remove extension ambiguity arising from use of the GROUP encoding instruction.

B.1. Example 1

Consider the following type definition:

```
SEQUENCE {
    one     [GROUP] SEQUENCE {
        two     UTF8String,
        ... -- Extension insertion point (I1).
    },
    three  INTEGER OPTIONAL,
    ... -- Extension insertion point (I2).
}
```

The associated grammar is:

```
P1:  S ::= one three I2
P2:  one ::= two I1
P3:  two ::= "two"
P4:  I1 ::= "*" I1
P5:  I1 ::=
P6:  three ::= "three"
P7:  three ::=
P8:  I2 ::= "*" I2
P9:  I2 ::=
```

This grammar leads to the following sets and predicates:

```
First(P4) = { "*" }
First(P5) = { }
Preselected(P4) = Preselected(P5) = Empty(P4) = false
Empty(P5) = true
Follow(I1) = { "three", "*", "$" }
Select(P4) = First(P4) = { "*" }
Select(P5) = First(P5) + Follow(I1) = { "three", "*", "$" }
```

```
   First(P6) = { "three" }
   First(P7) = { }
   Preselected(P6) = Preselected(P7) = Empty(P6) = false
   Empty(P7) = true
   Follow(three) = { "*", "$" }
   Select(P6) = First(P6) = { "three" }
   Select(P7) = First(P7) + Follow(three) = { "*", "$" }

   First(P8) = { "*" }
   First(P9) = { }
   Preselected(P8) = Preselected(P9) = Empty(P8) = false
   Empty(P9) = true
   Follow(I2) = { "$" }
   Select(P8) = First(P8) = { "*" }
   Select(P9) = First(P9) + Follow(I2) = { "$" }
```

The intersection of Select(P4) and Select(P5) is not empty; hence, the grammar is not deterministic, and the type definition is not valid. If an RXER decoder encounters an unrecognized element immediately after a <two> element, then it will not know whether to associate it with extension insertion point I1 or I2.

The non-determinism can be resolved with either a NO-INSERTIONS or HOLLOW-INSERTIONS encoding instruction. Consider this revised type definition:

```
   SEQUENCE {
       one    [GROUP] [HOLLOW-INSERTIONS] SEQUENCE {
           two    UTF8String,
           ... -- Extension insertion point (I1).
       },
       three  INTEGER OPTIONAL,
       ... -- Extension insertion point (I2).
   }
```

The associated grammar is:

```
   P1:  S ::= one three I2
   P10: one ::= two
   P3:  two ::= "two"
   P6:  three ::= "three"
   P7:  three ::=
   P8:  I2 ::= "*" I2
   P9:  I2 ::=
```

With the addition of the HOLLOW-INSERTIONS encoding instruction, the P4 and P5 productions are no longer generated, and the conflict between Select(P4) and Select(P5) no longer exists. The Select Sets

for P6, P7, P8, and P9 are unchanged. A decoder will now assume that
an unrecognized element is to be associated with extension insertion
point I2. It is still free to associate an unrecognized attribute
with either extension insertion point. If a NO-INSERTIONS encoding
instruction had been used, then an unrecognized attribute could only
be associated with extension insertion point I2.

The non-determinism could also be resolved by adding a NO-INSERTIONS
or HOLLOW-INSERTIONS encoding instruction to the outer SEQUENCE:

```
[HOLLOW-INSERTIONS] SEQUENCE {
    one    [GROUP] SEQUENCE {
        two    UTF8String,
        ... -- Extension insertion point (I1).
    },
    three  INTEGER OPTIONAL,
    ... -- Extension insertion point (I2).
}
```

The associated grammar is:

```
P11: S ::= one three
P2:  one ::= two I1
P3:  two ::= "two"
P4:  I1 ::= "*" I1
P5:  I1 ::=
P6:  three ::= "three"
P7:  three ::=
```

This grammar leads to the following sets and predicates:

```
First(P4) = { "*" }
First(P5) = { }
Preselected(P4) = Preselected(P5) = Empty(P4) = false
Empty(P5) = true
Follow(I1) = { "three", "$" }
Select(P4) = First(P4) = { "*" }
Select(P5) = First(P5) + Follow(I1) = { "three", "$" }

First(P6) = { "three" }
First(P7) = { }
Preselected(P6) = Preselected(P7) = Empty(P6) = false
Empty(P7) = true
Follow(three) = { "$" }
Select(P6) = First(P6) = { "three" }
Select(P7) = First(P7) + Follow(three) = { "$" }
```

The intersection of Select(P4) and Select(P5) is empty, as is the
intersection of Select(P6) and Select(P7); hence, the grammar is
deterministic, and the type definition is valid. A decoder will now
assume that an unrecognized element is to be associated with
extension insertion point I1. It is still free to associate an
unrecognized attribute with either extension insertion point. If a
NO-INSERTIONS encoding instruction had been used, then an
unrecognized attribute could only be associated with extension
insertion point I1.

B.2. Example 2

   Consider the following type definition:

```
    SEQUENCE {
        one  [GROUP] CHOICE {
            two  UTF8String,
            ... -- Extension insertion point (I1).
        } OPTIONAL
    }
```

   The associated grammar is:

```
    P1:  S ::= one
    P2:  one ::= two
    P3:  one ::= I1
    P4:  one ::=
    P5:  two ::= "two"
    P6:  I1 ::= "*" I1
    P7:  I1 ::=
```

   This grammar leads to the following sets and predicates:

```
    First(P2) = { "two" }
    First(P3) = { "*" }
    First(P4) = { }
    Preselected(P2) = Preselected(P3) = Preselected(P4) = false
    Empty(P2) = false
    Empty(P3) = Empty(P4) = true
    Follow(one) = { "$" }
    Select(P2) = First(P2) = { "two" }
    Select(P3) = First(P3) + Follow(one) = { "*", "$" }
    Select(P4) = First(P4) + Follow(one) = { "$" }

    First(P6) = { "*" }
    First(P7) = { }
    Preselected(P6) = Preselected(P7) = Empty(P6) = false
    Empty(P7) = true
```

```
      Follow(I1) = { "$" }
      Select(P6) = First(P6) = { "*" }
      Select(P7) = First(P7) + Follow(I1) = { "$" }
```

   The intersection of Select(P3) and Select(P4) is not empty; hence,
   the grammar is not deterministic, and the type definition is not
   valid. If the <two> element is not present, then a decoder cannot
   determine whether the "one" alternative is absent, or present with an
   unknown extension that generates no elements.

   The non-determinism can be resolved with either a
   SINGULAR-INSERTIONS, UNIFORM-INSERTIONS, or MULTIFORM-INSERTIONS
   encoding instruction. The MULTIFORM-INSERTIONS encoding instruction
   is the least restrictive. Consider this revised type definition:

```
      SEQUENCE {
          one  [GROUP] [MULTIFORM-INSERTIONS] CHOICE {
              two  UTF8String,
              ... -- Extension insertion point (I1).
          } OPTIONAL
      }
```

   The associated grammar is:

```
      P1:  S ::= one
      P2:  one ::= two
      P8:  one ::= "*" I1
      P4:  one ::=
      P5:  two ::= "two"
      P6:  I1 ::= "*" I1
      P7:  I1 ::=
```

   This grammar leads to the following sets and predicates:

```
      First(P2) = { "two" }
      First(P8) = { "*" }
      First(P4) = { }
      Preselected(P2) = Preselected(P8) = Preselected(P4) = false
      Empty(P2) = Empty(P8) = false
      Empty(P4) = true
      Follow(one) = { "$" }
      Select(P2) = First(P2) = { "two" }
      Select(P8) = First(P8) = { "*" }
      Select(P4) = First(P4) + Follow(one) = { "$" }

      First(P6) = { "*" }
      First(P7) = { }
      Preselected(P6) = Preselected(P7) = Empty(P6) = false
```

```
     Empty(P7) = true
     Follow(I1) = { "$" }
     Select(P6) = First(P6) = { "*" }
     Select(P7) = First(P7) + Follow(I1) = { "$" }
```

The intersection of Select(P2) and Select(P8) is empty, as is the
intersection of Select(P2) and Select(P4), the intersection of
Select(P8) and Select(P4), and the intersection of Select(P6) and
Select(P7); hence, the grammar is deterministic, and the type
definition is valid. A decoder will now assume the "one" alternative
is present if it sees at least one unrecognized element, and absent
otherwise.

B.3. Example 3

Consider the following type definition:

```
     SEQUENCE {
         one    [GROUP] CHOICE {
             two    UTF8String,
             ... -- Extension insertion point (I1).
         },
         three  [GROUP] CHOICE {
             four   UTF8String,
             ... -- Extension insertion point (I2).
         }
     }
```

The associated grammar is:

```
     P1:  S ::= one three
     P2:  one ::= two
     P3:  one ::= I1
     P4:  two ::= "two"
     P5:  I1 ::= "*" I1
     P6:  I1 ::=
     P7:  three ::= four
     P8:  three ::= I2
     P9:  four ::= "four"
     P10: I2 ::= "*" I2
     P11: I2 ::=
```

This grammar leads to the following sets and predicates:

```
     First(P2) = { "two" }
     First(P3) = { "*" }
     Preselected(P2) = Preselected(P3) = Empty(P2) = false
     Empty(P3) = true
```

```
   Follow(one) = { "four", "*", "$" }
   Select(P2) = First(P2) = { "two" }
   Select(P3) = First(P3) + Follow(one) = { "*", "four", "$" }

   First(P5) = { "*" }
   First(P6) = { }
   Preselected(P5) = Preselected(P6) = Empty(P5) = false
   Empty(P6) = true
   Follow(I1) = { "four", "*", "$" }
   Select(P5) = First(P5) = { "*" }
   Select(P6) = First(P6) + Follow(I1) = { "four", "*", "$" }

   First(P7) = { "four" }
   First(P8) = { "*" }
   Preselected(P7) = Preselected(P8) = Empty(P7) = false
   Empty(P8) = true
   Follow(three) = { "$" }
   Select(P7) = First(P7) = { "four" }
   Select(P8) = First(P8) + Follow(three) = { "*", "$" }

   First(P10) = { "*" }
   First(P11) = { }
   Preselected(P10) = Preselected(P11) = Empty(P10) = false
   Empty(P11) = true
   Follow(I2) = { "$" }
   Select(P10) = First(P10) = { "*" }
   Select(P11) = First(P11) + Follow(I2) = { "$" }
```

The intersection of Select(P5) and Select(P6) is not empty; hence, the grammar is not deterministic, and the type definition is not valid. If the first child element is an unrecognized element, then a decoder cannot determine whether to associate it with extension insertion point I1, or to associate it with extension insertion point I2 by assuming that the "one" component has an unknown extension that generates no elements.

The non-determinism can be resolved with either a SINGULAR-INSERTIONS or UNIFORM-INSERTIONS encoding instruction. Consider this revised type definition using the SINGULAR-INSERTIONS encoding instruction:

```
    SEQUENCE {
        one    [GROUP] [SINGULAR-INSERTIONS] CHOICE {
            two    UTF8String,
            ... -- Extension insertion point (I1).
        },
        three  [GROUP] CHOICE {
            four   UTF8String,
            ... -- Extension insertion point (I2).
        }
    }
```

The associated grammar is:

```
    P1:  S ::= one three
    P2:  one ::= two
    P12: one ::= "*"
    P4:  two ::= "two"
    P7:  three ::= four
    P8:  three ::= I2
    P9:  four ::= "four"
    P10: I2 ::= "*" I2
    P11: I2 ::=
```

With the addition of the SINGULAR-INSERTIONS encoding instruction,
the P5 and P6 productions are no longer generated. The grammar leads
to the following sets and predicates for the P2 and P12 productions:

```
    First(P2) = { "two" }
    First(P12) = { "*" }
    Preselected(P2) = Preselected(P12) = false
    Empty(P2) = Empty(P12) = false
    Follow(one) = { "four", "*", "$" }
    Select(P2) = First(P2) = { "two" }
    Select(P12) = First(P12) = { "*" }
```

The sets for P5 and P6 are no longer generated, and the remaining
sets are unchanged.

The intersection of Select(P2) and Select(P12) is empty, as is the
intersection of Select(P7) and Select(P8) and the intersection of
Select(P10) and Select(P11); hence, the grammar is deterministic, and
the type definition is valid. If the first child element is an
unrecognized element, then a decoder will now assume that it is
associated with extension insertion point I1. Whatever follows,
possibly including another unrecognized element, will belong to the
"three" component.

Now consider the type definition using the UNIFORM-INSERTIONS
encoding instruction instead:

```
SEQUENCE {
    one    [GROUP] [UNIFORM-INSERTIONS] CHOICE {
        two    UTF8String,
        ... -- Extension insertion point (I1).
    },
    three  [GROUP] CHOICE {
        four   UTF8String,
        ... -- Extension insertion point (I2).
    }
}
```

The associated grammar is:

```
P1:  S ::= one three
P2:  one ::= two
P13: one ::= "*"
P14: one ::= "*1" I1
P4:  two ::= "two"
P15: I1 ::= "*1" I1
P6:  I1 ::=
P7:  three ::= four
P8:  three ::= I2
P9:  four ::= "four"
P10: I2 ::= "*" I2
P11: I2 ::=
```

This grammar leads to the following sets and predicates for the P2,
P13, P14, P15, and P6 productions:

```
First(P2) = { "two" }
First(P13) = { "*" }
First(P14) = { "*1" }
Preselected(P2) = Preselected(P13) = Preselected(P14) = false
Empty(P2) = Empty(P13) = Empty(P14) = false
Follow(one) = { "four", "*", "$" }
Select(P2) = First(P2) = { "two" }
Select(P13) = First(P13) = { "*" }
Select(P14) = First(P14) = { "*1" }
```

```
   First(P15) = { "*1" }
   First(P6) = { }
   Preselected(P15) = Preselected(P6) = Empty(P15) = false
   Empty(P6) = true
   Follow(I1) = { "four", "*", "$" }
   Select(P15) = First(P15) = { "*1" }
   Select(P6) = First(P6) + Follow(I1) = { "four", "*", "$" }
```

The remaining sets are unchanged.

The intersection of Select(P2) and Select(P13) is empty, as is the
intersection of Select(P2) and Select(P14), the intersection of
Select(P13) and Select(P14) and the intersection of Select(P15) and
Select(P6); hence, the grammar is deterministic, and the type
definition is valid. If the first child element is an unrecognized
element, then a decoder will now assume that it and every subsequent
unrecognized element with the same name are associated with I1.
Whatever follows, possibly including another unrecognized element
with a different name, will belong to the "three" component.

A consequence of using the UNIFORM-INSERTIONS encoding instruction is
that any future extension to the "three" component will be required
to generate elements with names that are different from the names of
the elements generated by the "one" component. With the
SINGULAR-INSERTIONS encoding instruction, extensions to the "three"
component are permitted to generate elements with names that are the
same as the names of the elements generated by the "one" component.

B.4. Example 4

   Consider the following type definition:

```
   SEQUENCE OF one [GROUP] CHOICE {
       two    UTF8String,
       ... -- Extension insertion point (I1).
   }
```

   The associated grammar is:

```
   P1:  S ::= one S
   P2:  S ::=
   P3:  one ::= two
   P4:  one ::= I1
   P5:  two ::= "two"
   P6:  I1 ::= "*" I1
   P7:  I1 ::=
```

This grammar leads to the following sets and predicates:

```
First(P1) = { "two", "*" }
First(P2) = { }
Preselected(P1) = Preselected(P2) = false
Empty(P1) = Empty(P2) = true
Follow(S) = { "$" }
Select(P1) = First(P1) + Follow(S) = { "two", "*", "$" }
Select(P2) = First(P2) + Follow(S) = { "$" }

First(P3) = { "two" }
First(P4) = { "*" }
Preselected(P3) = Preselected(P4) = Empty(P3) = false
Empty(P4) = true
Follow(one) = { "two", "*", "$" }
Select(P3) = First(P3) = { "two" }
Select(P4) = First(P4) + Follow(one) = { "*", "two", "$" }

First(P6) = { "*" }
First(P7) = { }
Preselected(P6) = Preselected(P7) = Empty(P6) = false
Empty(P7) = true
Follow(I1) = { "two", "*", "$" }
Select(P6) = First(P6) = { "*" }
Select(P7) = First(P7) + Follow(I1) = { "two", "*", "$" }
```

The intersection of Select(P1) and Select(P2) is not empty, as is the
intersection of Select(P3) and Select(P4) and the intersection of
Select(P6) and Select(P7); hence, the grammar is not deterministic,
and the type definition is not valid. If a decoder encounters two or
more unrecognized elements in a row, then it cannot determine whether
this represents one instance or more than one instance of the "one"
component. Even without unrecognized elements, there is still a
problem that an encoding could contain an indeterminate number of
"one" components using an extension that generates no elements.

The non-determinism cannot be resolved with a UNIFORM-INSERTIONS
encoding instruction. Consider this revised type definition using
the UNIFORM-INSERTIONS encoding instruction:

```
SEQUENCE OF one [GROUP] [UNIFORM-INSERTIONS] CHOICE {
    two    UTF8String,
    ... -- Extension insertion point (I1).
}
```

The associated grammar is:

```
P1:  S ::= one S
P2:  S ::=
P3:  one ::= two
P8:  one ::= "*"
P9:  one ::= "*1" I1
P5:  two ::= "two"
P10: I1 ::= "*1" I1
P7:  I1 ::=
```

This grammar leads to the following sets and predicates:

```
First(P1) = { "two", "*", "*1" }
First(P2) = { }
Preselected(P1) = Preselected(P2) = Empty(P1) = false
Empty(P2) = true
Follow(S) = { "$" }
Select(P1) = First(P1) = { "two", "*", "*1" }
Select(P2) = First(P2) + Follow(S) = { "$" }

First(P3) = { "two" }
First(P8) = { "*" }
First(P9) = { "*1" }
Preselected(P3) = Preselected(P8) = Preselected(P9) = false
Empty(P3) = Empty(P8) = Empty(P9) = false
Follow(one) = { "two", "*", "*1", "$" }
Select(P3) = First(P3) = { "two" }
Select(P8) = First(P8) = { "*" }
Select(P9) = First(P9) = { "*1" }

First(P10) = { "*1" }
First(P7) = { }
Preselected(P10) = Preselected(P7) = Empty(P10) = false
Empty(P7) = true
Follow(I1) = { "two", "*", "*1", "$" }
Select(P10) = First(P10) = { "*1" }
Select(P7) = First(P7) + Follow(I1) = { "two", "*", "*1", "$" }
```

The intersection of Select(P1) and Select(P2) is now empty, but the
intersection of Select(P10) and Select(P7) is not; hence, the grammar
is not deterministic, and the type definition is not valid. The
problem of an indeterminate number of "one" components from an
extension that generates no elements has been solved. However, if a
decoder encounters a series of elements with the same name, it cannot
determine whether this represents one instance or more than one
instance of the "one" component.

The non-determinism can be fully resolved with a SINGULAR-INSERTIONS
encoding instruction. Consider this revised type definition:

```
SEQUENCE OF one [GROUP] [SINGULAR-INSERTIONS] CHOICE {
    two    UTF8String,
    ... -- Extension insertion point (I1).
}
```

The associated grammar is:

```
P1:  S ::= one S
P2:  S ::=
P3:  one ::= two
P8:  one ::= "*"
P5:  two ::= "two"
```

This grammar leads to the following sets and predicates:

```
First(P1) = { "two", "*" }
First(P2) = { }
Preselected(P1) = Preselected(P2) = Empty(P1) = false
Empty(P2) = true
Follow(S) = { "$" }
Select(P1) = First(P1) = { "two", "*" }
Select(P2) = First(P2) + Follow(S) = { "$" }

First(P3) = { "two" }
First(P8) = { "*" }
Preselected(P3) = Preselected(P8) = false
Empty(P3) = Empty(P8) = false
Follow(one) = { "two", "*", "$" }
Select(P3) = First(P3) = { "two" }
Select(P8) = First(P8) = { "*" }
```

The intersection of Select(P1) and Select(P2) is empty, as is the
intersection of Select(P3) and Select(P8); hence, the grammar is
deterministic, and the type definition is valid. A decoder now knows
that every extension to the "one" component will generate a single
element, so the correct number of "one" components will be decoded.

Appendix C. Extension and Versioning Examples

   This appendix is non-normative.

C.1. Valid Extensions for Insertion Encoding Instructions

   The first example shows extensions that satisfy the HOLLOW-INSERTIONS
   encoding instruction.

```
    [HOLLOW-INSERTIONS] CHOICE {
        one     BOOLEAN,
        ...,
        two     [ATTRIBUTE] INTEGER,
        three  [GROUP] SEQUENCE {
            four  [ATTRIBUTE] UTF8String,
            five  [ATTRIBUTE] INTEGER OPTIONAL,
            ...
        },
        six     [GROUP] CHOICE {
            seven  [ATTRIBUTE] BOOLEAN,
            eight  [ATTRIBUTE] INTEGER
        }
    }
```

   The "two" and "six" components generate only attributes.

   The "three" component in its current form does not generate elements.
   Any extension to the "three" component will need to do likewise to
   avoid breaking forward compatibility.

   The second example shows extensions that satisfy the
   SINGULAR-INSERTIONS encoding instruction.

```
    [SINGULAR-INSERTIONS] CHOICE {
        one     BOOLEAN,
        ...,
        two     INTEGER,
        three  [GROUP] SEQUENCE {
            four   [ATTRIBUTE] UTF8String,
            five   INTEGER
        },
        six     [GROUP] CHOICE {
            seven  BOOLEAN,
            eight  INTEGER
        }
    }
```

The "two" component will always generate a single <two> element.

The "three" component will always generate a single <five> element.
It will also generate a "four" attribute, but any number of
attributes is allowed by the SINGULAR-INSERTIONS encoding
instruction.

The "six" component will either generate a single <seven> element or
a single <eight> element. Either case will satisfy the requirement
that there will be a single element in any given encoding of the
extension.

The third example shows extensions that satisfy the
UNIFORM-INSERTIONS encoding instruction.

```
[UNIFORM-INSERTIONS] CHOICE {
    one     BOOLEAN,
    ...,
    two     INTEGER,
    three  [GROUP] SEQUENCE SIZE(1..MAX) OF four INTEGER,
    five   [GROUP] SEQUENCE {
        six    [ATTRIBUTE] UTF8String OPTIONAL,
        seven  INTEGER
    },
    eight  [GROUP] CHOICE {
        nine   BOOLEAN,
        ten    [GROUP] SEQUENCE SIZE(1..MAX) OF eleven INTEGER
    }
}
```

The "two" component will always generate a single <two> element.

The "three" component will always generate one or more <four>
elements.

The "five" component will always generate a single <seven> element.
It may also generate a "six" attribute, but any number of attributes
is allowed by the UNIFORM-INSERTIONS encoding instruction.

The "eight" component will either generate a single <nine> element or
one or more <eleven> elements. Either case will satisfy the
requirement that there must be one or more elements with the same
name in any given encoding of the extension.

C.2. Versioning Example

   Making extensions that are not forward compatible is permitted
   provided that the incompatibility is signalled with a version
   indicator attribute.

   Suppose that version 1.0 of a specification contains the following
   type definition:

```
      MyMessageType ::= SEQUENCE {
          version  [ATTRIBUTE] [VERSION-INDICATOR]
                      UTF8String ("1.0", ...) DEFAULT "1.0",
          one      [GROUP] [SINGULAR-INSERTIONS] CHOICE {
              two  BOOLEAN,
              ...
          },
          ...
      }
```

   An attribute is to be added to the CHOICE for version 1.1. This
   change is not forward compatible since it does not satisfy the
   SINGULAR-INSERTIONS encoding instruction. Therefore, the version
   indicator attribute must be updated at the same time (or added if it
   wasn't already present). This results in the following new type
   definition for version 1.1:

```
      MyMessageType ::= SEQUENCE {
          version  [ATTRIBUTE] [VERSION-INDICATOR]
                      UTF8String ("1.0", ..., "1.1") DEFAULT "1.0",
          one      [GROUP] [SINGULAR-INSERTIONS] CHOICE {
              two    BOOLEAN,
              ...,
              three  [ATTRIBUTE] INTEGER -- Added in Version 1.1
          },
          ...
      }
```

   If a version 1.1 conformant application hasn't used the version 1.1
   extension in a value of MyMessageType, then it is allowed to set the
   value of the version attribute to "1.0".

   A pair of elements is added to the CHOICE for version 1.2. Again the
   change does not satisfy the SINGULAR-INSERTIONS encoding instruction.
   The type definition for version 1.2 is:

```
    MyMessageType ::= SEQUENCE {
       version  [ATTRIBUTE] [VERSION-INDICATOR]
                   UTF8String ("1.0", ..., "1.1" | "1.2")
                      DEFAULT "1.0",
       one       [GROUP] [SINGULAR-INSERTIONS] CHOICE {
           two     BOOLEAN,
           ...,
           three  [ATTRIBUTE] INTEGER, -- Added in Version 1.1
           four   [GROUP] SEQUENCE {
               five  UTF8String,
               six   GeneralizedTime
           } -- Added in version 1.2
       },
       ...
    }
```

If a version 1.2 conformant application hasn't used the version 1.2
extension in a value of MyMessageType, then it is allowed to set the
value of the version attribute to "1.1". If it hasn't used either of
the extensions, then it is allowed to set the value of the version
attribute to "1.0".

Author's Address

   Dr. Steven Legg
   eB2Bcom
   Suite 3, Woodhouse Corporate Centre
   935 Station Street
   Box Hill North, Victoria 3129
   AUSTRALIA

   Phone: +61 3 9896 7830
   Fax: +61 3 9896 7801
   EMail: steven.legg@eb2bcom.com

Full Copyright Statement

Intellectual Property

Acknowledgement