



Mapping an ASN.1 Schema to an Apache Avro Schema

Objective Systems, Inc.

September 29, 2026

Copyright Notice

Copyright ©2026 Objective Systems, Inc. All rights reserved.

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information

Comments, suggestions, and inquiries regarding this document may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

1	Introduction to the Mapping.....	1
1.1	Avro Syntax.....	1
1.2	Indecipherable Data.....	1
1.3	Version Compatibility.....	1
1.4	Name Mutation.....	1
1.5	User Inputs.....	2
2	Avro Schema Generation.....	3
2.1	General Procedures.....	3
2.1.1	Procedure To Make Unique Name.....	3
2.1.2	Assigning Avro Type Name and Namespace for T.....	3
2.2	Procedures to Generate Avro Type from ASN.1 Type.....	5
2.2.1	From BOOLEAN.....	5
2.2.2	From INTEGER.....	5
2.2.3	From ENUMERATED.....	6
2.2.3.1	Examples.....	6
2.2.4	From BIT STRING.....	7
2.2.5	From OCTET STRING.....	7
2.2.5.1	Examples.....	8
2.2.6	From NULL.....	8
2.2.7	From DefinedType.....	8
2.2.8	From SEQUENCE.....	10
2.2.8.1	Example.....	11
2.2.9	From SEQUENCE OF T.....	12
2.2.9.1	Example.....	12
2.2.10	From SET and SET OF.....	14
2.2.11	From CHOICE.....	14
2.2.11.1	Procedure to Add T to a union.....	14
2.2.11.2	Examples.....	15
2.2.12	OBJECT IDENTIFIER.....	17
2.2.13	Unsupported Types.....	17
2.2.14	Character Strings.....	17
2.2.15	Open Type.....	17
2.2.15.1	Example.....	18
2.2.16	Parameterized Types.....	20

1 Introduction to the Mapping

Objective System's ASN.1 compiler (asn1c) can generate Apache Avro schemas, and Avro encoding functions, from an ASN.1 schema. This document describes how asn1c maps an ASN.1 schema to Avro.

1.1 Avro Syntax

The mapping presented here produces an Avro schema in JSON format. It was based on the specification for Avro 1.12.0.

1.2 Indecipherable Data

In those cases where an ASN.1 decoder might encounter blocks of data that it cannot decode, the Avro schema does not provide a way to carry the unknown ASN.1 data; such data is thus forced to be discarded when converting from an ASN.1 encoding to an Avro encoding. This applies to:

1. unknown extension additions in a CHOICE type
2. unknown extension additions in a SEQUENCE type
3. unknown actual type for an open type

1.3 Version Compatibility

The Avro schemas generated from two different versions of the same ASN.1 schema will, in general, NOT have compatible encodings. In particular, compatibility breaks when:

1. additions are made to an extensible CHOICE
2. additions are made to an extensible object set

1.4 Name Mutation

Though arguably unlikely, it is possible that the Avro schema generated from one version of an ASN.1 schema will use different names for the same ASN.1 entity than the schema generated from a different version of the ASN.1. Fundamentally, this is a consequence of converting ASN.1 names to Avro names (which follow a different format): the conversion procedure can produce name conflicts that were not in the ASN.1; it must subsequently resolve those conflicts. When such conflicts do occur, rearranging the ASN.1 schema text can change the resolution and thus the final, generated name. Likewise, editing an ASN.1 schema could introduce new conflicts, thus altering the final, generated name. Fortunately, altering names does not alter the Avro binary encoding, but it will alter code generated from the Avro schema.

Conflict example: A type named "Abc-xyz" and a component named "xyz" in type "Abc" would both use Avro type name "Abc_xyz", so one of them must use a different name.

1.5 User Inputs

User inputs can play a role in the mapping. In particular, configuration of the C integer representation for INTEGER types can affect the choice of the Avro type.

2 Avro Schema Generation

An Avro schema in JSON format contains a single, top-level schema definition. Any other schema definitions required by it are nested inside it, and any schema definitions required by a nested definition is likewise nested inside that nested definition, and so on. This means that a recursive algorithm is the most natural way to describe how to generate an Avro schema from an ASN.1 type, and so that is what we provide here.

Schema generation begins by generating an Avro type for some type T where T is the Type in some TypeAssignment. For example, we might begin by generating a type for the SEQUENCE in the following TypeAssignment (which assigns the name MyPDU to the given SEQUENCE):

```
MyPDU ::= SEQUENCE { ... }
```

Schema generation then proceeds recursively.

2.1 General Procedures

2.1.1 Procedure To Make Unique Name

This procedure is invoked to ensure the uniqueness of a generated name. It assumes a known set of names that have already been generated, to which the new name will be added.

The procedure to make a given name unique within a set of names is as follows.

```
name = name to make unique
nameSet = set of names the new name will be added to

if name is already in nameSet
    x = smallest integer >= 2 such that <name>_x is not in nameSet
    name = <name>_x
```

2.1.2 Assigning Avro Type Name and Namespace for T

This procedure is invoked to assign a name and namespace to an ASN.1 type, T , when T is represented as a named type in Avro. This procedure invokes itself recursively, when it needs a name for an enclosing type, for use as a basis for T 's name; this includes invoking it recursively for ASN.1 types that are represented as an array or union, where the assigned name will not itself be used in the Avro schema.

This procedure includes a step for ensuring that names are unique. This means that the processing order can impact the outcome; therefore, this procedure should only be invoked when called for by another procedure.

We use ASN.1 module names for Avro namespace names (with hyphen-to-underscore conversion), thereby providing a corresponding namespace in Avro. Note: It will not always be necessary to

generate the given namespace value in the Avro schema, as it will frequently be the case that the desired namespace will be inherited from the enclosing schema anyway.

The steps for the procedure are as follows:

- 1 If the ASN.1 provides a name for T, (i.e. T is the type in a type assignment, such as
SomeName ::= T):
 - 1.1 name = the ASN.1 name provided for T; namespace = the ASN.1 module name
- 2 Otherwise, if T is a SEQUENCE OF X and X is represented using an Avro primitive type:
 - 2.1 name = "SeqOf_<primitive name>"; namespace = "asn1_builtin". This completes the procedure (there is no need to remove hyphens or make the name unique).
- 3 Otherwise, if T is the component type in a SEQUENCE OF T:
 - 3.1 Let seqofName be the name assigned to the SEQUENCE OF (invoke this procedure, if a name is not yet assigned).
 - 3.2 Let repName be "item" if the SEQUENCE OF was defined without a component identifier; otherwise, the component identifier.
 - 3.3 name = "<seqofName>_<repName>"; namespace = T's ASN.1 module name.
- 4 Otherwise, if T is a component type in a SEQUENCE, OR an alternative in a CHOICE:
 - 4.1 Let parentName be the name assigned to the SEQUENCE/CHOICE type (invoke this procedure, if a name is not yet assigned)
 - 4.2 name = "<parentName>_<component/alternative's ASN.1 name>"; namespace = T's ASN.1 module name.
- 5 Otherwise, if T is the type in a BIT STRING (CONTAINING T) or an OCTET STRING (CONTAINING T):
 - 5.1 Use the name and namespace assigned to the BIT/OCTET STRING (invoke this procedure, if a name is not yet assigned); this completes the procedure (name is already unique).
- 6 Otherwise, if T is one of the actual types for an open type, taken from an information object:
 - 6.1 Let openName be the name assigned to the open type (invoke this procedure, if a name is not yet assigned).
 - 6.2 Let altName be one of the following (use the first one that exists): 1) the ASN.1 name of the key value for the information object, 2) the ASN.1 name of the information object itself, 3) the generic ASN.1 name for the type (use "SEQUENCE", "SEQUENCE-OF", etc. for constructed types).
 - 6.3 name = "<openName>_<altName>"; namespace = T's ASN.1 module name.

- 7 Convert any hyphen in name or namespace to an underscore.
- 8 Make name unique for the set of names in namespace, following *2.1.1 Procedure To Make Unique Name*.

2.2 Procedures to Generate Avro Type from ASN.1 Type

This section describes the procedures for generating an Avro type from an ASN.1 type.

There are two cases where these procedures *may* generate a reference to a named type (previously generated, or being generated):

1. DefinedType (i.e. a reference to a named type in the ASN.1)
2. (unnamed) SEQUENCE OF (because some cases, e.g. SEQUENCE OF INTEGER, produce a reusable type)

Only the procedure for SEQUENCE OF can (possibly) generate an array. In certain contexts, the procedures disallow use of an array type and require a record type instead.

Whenever these procedures generate a named type, they determine the name and namespace by invoking *2.1.2 Assigning Avro Type Name and Namespace for T*.

2.2.1 From BOOLEAN

Use boolean.

2.2.2 From INTEGER

The Avro representation depends on type constraints, according to the following logic:

- 1 If the type is configured as big integer, then use bytes.
- 2 Otherwise, if there is a non-extensible constraint, with finite lower and upper bounds falling within the range of an Avro int or an Avro long, then use int/long.¹
- 3 Otherwise (there are ASN.1 values that don't fit in an Avro long), then:
 - 3.1 The Avro type is chosen based on the C type used by asn1c to represent the INTEGER type. When an Avro long does not fit all values, the only C type which might fit all values is uint64, but since the C type is configurable, this clause covers all of the possible C types.
 - 3.2 If the C type is uint64, use bytes.
 - 3.3 Otherwise, if the C type is uint32 or int64, use long.
 - 3.4 Otherwise (uint16, uint8, int32, int16, or int8), use int.

¹ Note: in this case, you cannot simply map the C type to the Avro type. For example, INTEGER(0..65536) will map to C type OSUINT32 but will map to Avro int and not Avro long, which is what OSUINT32 would map to.

When a bytes representation is used, it carries the value in 2's complement form, in the minimal number of bytes, in big-endian format.

2.2.3 From ENUMERATED

The Avro representation is an enum.

Use the following steps to generate the enum:

- 1 If the type has not already been assigned a name and namespace, invoke *2.1.2 Assigning Avro Type Name and Namespace for T*.
- 2 If the ENUMERATED is extensible, generate a symbol, "UNKNOWN", which shall be specified as the default for the enum.
- 3 Generate a symbol for each root value, using the ASN.1 identifier name (see below).
- 4 Generate a symbol for each extension value, using the ASN.1 identifier name (see below).
- 5 A symbol is created from an ASN.1 identifier name as follows:
 - 5.1 Convert every hyphen to an underscore. The result will be a unique name, due to the uniqueness of the ASN.1 identifier (which cannot contain an underscore).

Notes:

- "UNKNOWN", being all uppercase, will not conflict with the ENUMERATED identifier names. This symbol is used to represent an unknown extension value (such as may be encountered by an ASN.1 decoder).

2.2.3.1 Examples

ASN.1:

```
Colors ::= ENUMERATED { red, blue }
```

Avro:

```
{ "type" : "enum",  
  "name" : "Colors",  
  "symbols" : ["red", "blue"]  
}
```

ASN.1:

```
Colors ::= ENUMERATED { red, blue, ... }
```

Avro:

```
{ "type" : "enum",  
  "name" : "Colors",  
  "symbols" : ["UNKNOWN", "red", "blue"]  
}
```

2.2.4 From BIT STRING

The representation varies, according to the following rules:

- 1 If the BIT STRING is a BIT STRING (CONTAINING T), then it is treated as if it had been simply T, with T being treated as if it had appeared where the BIT STRING had appeared (e.g. in a type assignment, or contained in another type). Use the type produced by invoking the applicable procedure to generate an Avro type from T; when doing so, if the BIT STRING was assigned a name by a type assignment, then treat T as having been assigned that name instead.
- 2 Otherwise, if the BIT STRING has a non-extensible size constraint which fixes the length to a multiple of 8 bits, then:
 - 2.1 If the type has not already been assigned a name and namespace, invoke *2.1.2 Assigning Avro Type Name and Namespace for T*.
 - 2.2 Use fixed with the assigned name and namespace, and a size equal to the fixed bit string length, divided by eight.
- 3 Otherwise, use the record type given below. If the record has already been generated, generate a reference to it, otherwise, generate the record.

```
{
  "type" : "record",
  "name" : "Asn1BitString",
  "namespace" : "asn1_builtin",
  "fields" : [
    { "name" : "unused_bits",
      "type" : "int"
    },
    { "name" : "data",
      "type" : "bytes"
    }
  ]
}
```

2.2.5 From OCTET STRING

The representation varies, according to the following rules:

- 1 If the OCTET STRING is an OCTET STRING (CONTAINING T), it will be treated the same as the corresponding BIT STRING case.
- 2 Otherwise, if the OCTET STRING has a non-extensible size constraint which fixes the length to k bytes, then:
 - 2.1 If the type has not already been assigned a name and namespace, invoke *2.1.2 Assigning Avro Type Name and Namespace for T*.
 - 2.2 Use fixed with the assigned name and namespace, and a size equal to k.

3 Otherwise, use bytes.

2.2.5.1 *Examples*

ASN.1:

```
FixedOcts ::= OCTET STRING (SIZE(12))
```

Avro:

```
{ "type" : "fixed",  
  "name" : "FixedOcts",  
  "size" : 12  
}
```

ASN.1:

```
OCTET STRING (SIZE(1..12))
```

Avro:

bytes

2.2.6 From NULL

Use null.

2.2.7 From DefinedType

In ASN.1, a DefinedType is a reference to a type name which was created by a type assignment. In the following example, the name in **bold** is a DefinedType:

```
MyPDU ::= SEQUENCE {  
    elem1 Color, -- a DefinedType, a reference to Color  
    elem2 SEQUENCE { ... }  
}  
  
-- Color is defined as a type by this type assignment:  
Color ::= ENUMERATED { red, green, blue }
```

When we generate a type for a DefinedType, we will do one of the following:

- generate a reference to an Avro primitive type (boolean, bytes, string, etc.)
- generate a reference to a named Avro type, which has already been (or is being) generated
- generate a new Avro type, according to the DefinedType's underlying ASN.1 type

The representation for a DefinedType is the representation for the type referenced by the DefinedType. The context matters for this: SEQUENCE OF has an array representation except in certain contexts, where it has a record representation.

The first time we encounter the DefinedType in a context where the representation is a named type, we will generate the named type; thereafter we will generate a reference to the named type.

The procedure is:

1. Locate the type assignment that defines the `DefinedType`. Let `Module` be the module that defines the type assignment, let `NewName` be the name of the `DefinedType`, and let `Type` represent the type assigned to `NewName`, as in:
`NewName ::= Type`
2. If `Type` is itself a `DefinedType` (`NewName` is an alias), then recursively invoke the procedure to generate a type for `Type`, thus completing the procedure. (This means references to `MyName` will be treated as references to `Type`, for example, to `SomeOtherName`.)
3. If the representation is a named type AND we have already generated (or begun generation of) `NewName` in namespace `Module`, then generate a reference to it: `"MyName"` or `"Module.MyName"` (qualify it if necessary), thus completing the procedure. Note: It may happen that a reference to `X` is generated within the definition of `X` (this will happen if the ASN.1 type is recursively defined).
4. If none of the above applied, then invoke the applicable procedure to generate a type for `Type`. (If the representation is a named type, this will generate a type named `MyName` in namespace `Module`.)

Example ASN.1:

```
MyIntType ::= INTEGER (0..10)
```

```
MyRecord ::= SEQUENCE {
    field1 INTEGER(0..10),
    field2 HelloRecord,
    field3 HelloRecord,
    field4 MyIntType
}
```

```
HelloRecord ::= SEQUENCE {
    okay BOOLEAN,
    recur MyRecord OPTIONAL
}
```

Example Avro, `MyRecord` as the main schema

```
(1) { "type" : "record",
      "name" : "MyRecord",
      "namespace" : "MyModule",
      "fields" : [
        { "name" : "field1",
          "type" : "int"
        },
        { "name" : "field2",
          "type" :
            { "type" : "record",
              "name" : "HelloRecord",
              "fields" : [
                { "name" : "okay",
```

```

        "type" : "boolean"
      },
      { "name" : "recur",
        "type" : ["null", "MyRecord"]
      }
    ]
  },
  { "name" : "field3",
    "type" : "HelloRecord"
  },
  { "name" : "field4",
    "type" : "int"
  }
]
}

```

- (1) Defining a record type at the point of its first usage.
- (2) Referring to a previously defined type.
- (3) Referring to a previously defined type.
- (4) MyIntType has primitive representation. A named type is not created.

2.2.8 From SEQUENCE

The Avro representation is a record. Use the following steps to generate it.

- 1 If the SEQUENCE has not already been assigned a name and namespace, invoke *2.1.2 Assigning Avro Type Name and Namespace for T*.
- 2 Generate a field for each component, in schema order, with root components first, treating the components inside an extension addition group as if they were not inside a group. The rules to generate a field are as follows:
 - 2.1 The field name based on the ASN.1 component name as follows:
 - 2.1.1 convert every hyphen to an underscore
 - 2.1.2 make the name unique among the field names by following *2.1.1 Procedure To Make Unique Name*.
 - 2.2 If the component's type's representation is null and the component is OPTIONAL or an extension addition, then:
 - 2.2.1 The field name is based on the ASN.1 component name as follows:
 - 2.2.1.1 convert every hyphen to an underscore
 - 2.2.1.2 append "_present"
 - 2.2.1.3 make the name unique among the field names by following *2.1.1 Procedure To Make Unique Name*.

2.2.2 The field type is boolean.

2.2.3 If the component is an extension addition, then the field shall have a default value of false.

2.3 Otherwise, if the component is OPTIONAL or an extension addition then:

2.3.1 The field type is a union. The first member of the union is "null". The second member is generated by invoking the applicable procedure to generate a type from the component's type. (Note: it is okay to use an array type for the second member.)

2.3.2 If the component is an extension addition, then the field shall have a default value of "null".

2.4 Otherwise, to generate the type, invoke the applicable procedure to generate a type from the component's type.

2.2.8.1 **Example**

ASN.1 (module name is "MyModule"):

```
PDU ::= SEQUENCE {  
    field1 BOOLEAN OPTIONAL,  
    field2 BOOLEAN,  
    field3 NULL OPTIONAL,  
    ...,  
    fieldext BOOLEAN,  
    fieldext-null NULL OPTIONAL,  
    ...,  
    field4 BOOLEAN  
}
```

```
{ "type" : "record",  
  "name" : "PDU",  
  "namespace" : "MyModule",  
  "fields" :  
  [  
(1)    { "name" : "field1", "type" : ["null", "boolean"] },  
        { "name" : "field2", "type" : "boolean" },  
(2)    { "name" : "field3_present", "type" : "boolean" },  
(3)    { "name" : "field4", "type" : "boolean" },  
(4)    { "name" : "fieldext", "type" : ["null", "boolean"],  
          "default" : "null" },  
(5)    { "name" : "fieldext_null_present", "type" : "boolean",  
          "default" : "false" }  
  ]  
}
```

(1) OPTIONAL component using union. There is no default value (it is a root component).

(2) OPTIONAL root component with NULL type. No default value.

(3) Notice the reordering: the root components come before the extension additions.

(4) OPTIONAL extension using union. It has a default of null.

(5) OPTIONAL extension with NULL type. It has a default of false.

2.2.9 From SEQUENCE OF T

If the context in which this procedure is invoked has not disallowed an array representation, then the representation is an array; otherwise, the representation is a record.

The procedure follows these steps:

- 1 If the type has not already been assigned a name and namespace, assign them according to 2.1.2 *Assigning Avro Type Name and Namespace for T*.
- 2 If the representation is an array type, generate an array type:
 - 2.1 If T 's representation is a named type AND the named type has already been (or is being) generated, then the array's `items` type shall be a reference to the assigned name and namespace. (Note: if T 's representation is a named type and T has already been assigned a name, then the named type will already have been, or is being, generated.²)
 - 2.2 Otherwise, generate the `items` type by invoking the applicable procedure to generate a type from T .
- 3 If the representation is a record type then:
 - 3.1 As a special case, if T is represented using an Avro primitive type, then the record type for the SEQUENCE OF might already have been generated. If so, then generate a reference to that name, completing this procedure.
 - 3.2 Generate a record type having the assigned name, with a single field, named "items". The type for the "items" field is an array, generated as in step 2 above.

2.2.9.1 Example

ASN.1 (Module name = "MyModule")

```
ListInts ::= SEQUENCE OF INTEGER
```

```
ListFlags ::= SEQUENCE OF SEQUENCE {  
    flag1 BOOLEAN,  
    flag2 BOOLEAN  
}
```

```
PDU ::= SEQUENCE {  
    list-ints ListInts,  
    list-flags ListFlags,  
}
```

- 2 Proof: SEQUENCE OF is the only case where we might assign a name to a type, not use the name (because we opted for an array representation), and then later want to use the name. So, suppose that T is a SEQUENCE OF (so that it's possible to have been assigned a name without having generated the named type). Then, in the present context, T would have an array representation, not a named representation as assumed. Therefore, it is not possible for T 's representation to be a named type, for T to have already been assigned a name, and for that named type to not have already been generated (or in process of being generated).

```

second-flags ListFlags,
sizes SEQUENCE OF ENUMERATED { big, small },
}

```

Avro (PDU used as outermost)

```

{ "type" : "record",
  "name" : "PDU",
  "namespace" : "MyModule",
  "fields" :
  [
    (1) { "name" : "list_ints",
          "type" : { "type" : "array", "items" : "long" }
        },
    { "name" : "list_flags",
      "type" :
        { "type" : "array",
          "items" :
            (2) { "type" : "record",
                  "name" : "ListFlags_item",
                  "fields" :
                    [
                      { "name" : "flag1", "type" : "boolean" },
                      { "name" : "flag2", "type" : "boolean" }
                    ]
                }
          }
        },
    (3) { "name" : "second_flags",
          "type" : { "type" : "array", "items" : "ListFlags_item" },
        },
    { "name" : "sizes",
      "type" :
        (4) { "type" : "array",
              "items" :
                { "type" : "enum",
                  "name" : "PDU_sizes_item",
                  "symbols" : ["big", "small"]
                }
              }
        },
  ]
}

```

- (1) The array type on this line would appear anywhere ListInts is referenced, except in a CHOICE. In this case, the repeated type is not represented using a named type.
- (2) This type (ListFlags_item) is generated for the anonymous SEQUENCE in ListFlags. It is referenced later, when ListFlags is referenced again.
- (3) Here, ListFlags has appeared again, and we reuse the previously generated item type, ListFlags_item.

- (4) This type is generated for the anonymous `ENUMERATED` on component sizes. Unlike the other cases, `PDU_sizes_item` will never be referenced; it only has a name because it must.

2.2.10 From SET and SET OF

We didn't find any usage of `SET` or `SET OF` in *AP schemas, but there seems to be no reason to not treat it the same as `SEQUENCE` and `SEQUENCE OF`.

2.2.11 From CHOICE

The Avro representation is a record. Generate the record as follows.

- 1 If the type has not already been assigned a name and namespace, invoke *2.1.2 Assigning Avro Type Name and Namespace for T*.
- 2 Generate a field named "selection". The type for this field is an enum, as follows:
 - 2.1 The enum's name shall be "<record name>_selection", with *2.1.1 Procedure To Make Unique Name* applied. Its namespace shall be the namespace of the record.
 - 2.2 If the `CHOICE` is extensible, add symbol "UNKNOWN" to the enum.
 - 2.3 For each alternative in the `CHOICE`, add a symbol to the enum based on the ASN.1 identifier for the alternative as follows:
 - 2.3.1 convert every hyphen to underscore. The result will be a unique name, due to the uniqueness of the ASN.1 identifier (which cannot contain an underscore).
- 3 Generate a field named "value". The type for this field is a union, as follows:
 - 3.1 If the `CHOICE` is extensible, add "null" to the union.
 - 3.2 For each alternative type (order does not matter), invoke *2.2.11.1 Procedure to Add T to a union*.

2.2.11.1 Procedure to Add T to a union

This procedure adds a type for ASN.1 type `T` to a given union, if it has not already been added.

- 1 Determine the Avro representation for `T`; disallow an array representation.
- 2 If `T`'s representation is a named type, and `T` has not yet been assigned a name and namespace, assign a name and namespace by following *2.1.2 Assigning Avro Type Name and Namespace for T*.
- 3 If `T`'s representation is a primitive or named type that has not been added to the union already, add it to the union by invoking the applicable procedure to generate an Avro type from `T` (this might simply generate a name or it might generate a named type).

- 4 If τ is an open type (its representation is a union), then invoke the procedure to generate a type for the open type, but with this one difference: instead of creating a new union, add the member types to this union.

2.2.11.2 Examples

ASN.1 (Module name is "MyModule")

```
MyType ::= SEQUENCE { /* fields elided */ }
```

```
MyChoice ::= {  
  first NULL,  
  second INTEGER,  
  third MyType,  
  fourth MyType,  
  x INTEGER  
}
```

Avro (using MyChoice)

```
{ "type" : "record",  
  "name" : "MyChoice",  
  "namespace" : "MyModule",  
  "fields" :  
  [  
    { "name" : "selection",  
      "type" :  
      {  
        "type" : "enum",  
        "name" : "MyChoice_selection"  
(1)      "symbols" : ["first", "second", "third", "fourth", "x"]  
      }  
    },  
(2)    { "name" : "value",  
          "type" : ["null", "long",  
                   { "type" : "record",  
                     "name" : "MyType",  
                     "fields" : [ /* fields elided */ ]  
                   }  
                 ]  
        }  
    ]  
  ]  
}
```

(1) One symbol in the enum for each alternative.

(2) There are five alternatives, but only three types in the union because some alternatives use the same Avro type (long and MyType have 2 alternatives each).

ASN.1 (Module name is "MyModule")

```
PDU ::= SEQUENCE {  
    mycho MyChoice  
}
```

```
MyChoice ::= {  
    first BOOLEAN,  
    second INTEGER,  
    third PDU,  
    ...  
}
```

Avro (using PDU)

```
{ "type" : "record",  
  "name" : "PDU",  
  "namespace" : "MyModule",  
  "fields" :  
  [  
    { "name" : "mycho",  
      "type" :  
      {  
        "type" : "record",  
        "name" : "MyChoice",  
        "namespace" : "MyModule",  
        "fields" :  
        [  
          { "name" : "selection",  
            "type" :  
            {  
              "type" : "enum",  
              "name" : "MyChoice_selection"  
(1)          "symbols" : ["UNKNOWN", "first", "second", "third"]  
            },  
          },  
(2)      { "name" : "value",  
            "type" : ["null", "boolean", "long", "PDU"]  
          }  
        ]  
      }  
    }  
  ]  
}
```

(1) Extensibility produces "UNKNOWN" as an alternative.

(2) Extensibility produces the null option. When the reference to "PDU" is generated, it is a type currently being generated.

2.2.12 OBJECT IDENTIFIER

If the context allows an array type, then use an array type with an `items` type of `long`. Otherwise, use the following record type:

```
{
  "type" : "record",
  "name" : "oid",
  "namespace" : "asn1_builtin",
  "fields" : [
    { "name" : "value",
      "type" : { "type" : "array", "items" : "int" }
    }
  ]
}
```

2.2.13 Unsupported Types

The following are NOT SUPPORTED:

- REAL
- RELATIVE-OID
- OID-IRI
- RELATIVE-OID-IRI
- EMBEDDED PDV
- EXTERNAL
- TIME (and derivatives)
- CHARACTER STRING (*unrestricted* character string type)

We did not find uses of these types in the *AP specifications that we looked at.

2.2.14 Character Strings

Use `string`.

2.2.15 Open Type

An unconstrained open type is represented using `null`. It has no encoding.

A constrained open type is represented as a union type, but when this procedure is invoked by the procedure for a `CHOICE` type, this procedure will instead add members to the union for the `CHOICE`.

Note that the open type representation is different from a `CHOICE` representation, which uses a record type with a field to carry an indication of which alternative was chosen.

The members of the union consist of the set of Avro types used to represent all of the ASN.1 actual types allowed by the constraining object set. For each actual type, we must know which member of the

union it corresponds to, but the order of the members is otherwise unimportant and unspecified³. The member types will not necessarily be named types (i.e. record, enum, or fixed).

Follow these steps:

1. An array representation is not allowed for the members of the union.
2. Generate a union if not given a union to generate members for.
3. If the constraining object set is extensible, add null to the union, if it has not already been added. (A null will be encoded when the actual type is unknown.)
4. For each type, T , allowed for the open type by the table constraint, invoke 2.2.11.1 *Procedure to Add T to a union*.

2.2.15.1 **Example**

Example (extracted from E1AP; the open type is shown in **bold**):

```
/* The ASN.1 */
ProtocolIE-ID ::= INTEGER (0..65535)

Criticality ::= ENUMERATED { reject, ignore, notify }

E1AP-PROTOCOL-IES ::= CLASS {
    &id                ProtocolIE-ID UNIQUE,
    &criticality        Criticality,
    &Value,
    &presence           Presence
}
WITH SYNTAX {
    ID                &id
    CRITICALITY        &criticality
    TYPE                &Value
    PRESENCE           &presence
}

ResourceStatusRequest ::= SEQUENCE {
    protocolIEs ProtocolIE-Container { { ResourceStatusRequestIES } },
    ...
}

ProtocolIE-Container { E1AP-PROTOCOL-IES : IESSetParam} ::=
    SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IESSetParam}}

ProtocolIE-Field { E1AP-PROTOCOL-IES : IESSetParam} ::= SEQUENCE {
    id E1AP-PROTOCOL-IES.&id ({IESSetParam}),
    criticality E1AP-PROTOCOL-IES.&criticality ({IESSetParam}{@id}),
    value E1AP-PROTOCOL-IES.&Value ({IESSetParam}{@id})
}

ResourceStatusRequestIES E1AP-PROTOCOL-IES ::= {
    { ID id-TransactionID CRITICALITY reject TYPE TransactionID
```

3 The order doesn't much matter because, as noted elsewhere, there is inherently a compatibility issue between schema versions when it comes to unions.

```

    PRESENCE mandatory} |
    { ID id-gNB-CU-CP-Measurement-ID CRITICALITY reject
      TYPE INTEGER (1..4095, ...) PRESENCE mandatory} |
    { ID id-gNB-CU-UP-Measurement-ID CRITICALITY ignore
      TYPE INTEGER (1..4095, ...) PRESENCE conditional} |
    { ID id-RegistrationRequest CRITICALITY reject TYPE RegistrationRequest
      PRESENCE mandatory}|
    { ID id-ReportCharacteristics CRITICALITY reject TYPE ReportCharacteristics
      PRESENCE conditional}
    { ID id-ReportingPeriodicity CRITICALITY ignore TYPE ReportingPeriodicity
      PRESENCE optional},
    ...
}

```

/* The Avro (if ResourceStatusRequest were the PDU).
 The actual type provided by the first three objects in the object set are
 represented using long (assuming the C representation is uint32 or uint64, which I
 believe would be the case as they are extensible types).
 */

```

{
  "type" : "record",
  "name" : "ResourceStatusRequest",
  "namespace" : "E1AP_PDU_Contents",
  "fields" :
  [
    {
      "name" : "protocolIEs",
      "type" :
      {
        "type" : "array",
        "items" :
        {
          "type" : "record",
          "name" : "ResourceStatusRequest_protocolIEs_item",
          "fields" :
          [
            {
              "name" : "id",
              "type" : "int"
            },
            {
              "name" : "criticality",
              "type" :
              {
                "type" : "enum",
                "name" : "Criticality",
                "namespace" : "E1AP_CommonDataTypes",
                "symbols" : ["reject", "ignore", "notify"]
              },
            },
            {
              "name" : "value",
              "type" :
              [
                "null",
                "long",

```

