ETSI ECN Tutorial (Version 1.1)

by

John Larmouth et al

(Version 1 was produced by John Larmouth, a member of ETSI STF 169; others have made later contributions to the work.)

NOTE — This tutorial represents work in progress. A recommended expiry date is Spring 2001, when users are strongly recommended to discard this version and seek a later one. See http://asn1.elibel.tm.fr/ecn/tutorial.htm

CONTENTS

1 Questions and Answers

1.1 What is Encoding Control Notation (ECN)?

ECN is a formally-defined notation (supported by tools) which, when applied to an ASN.1 abstract syntax specification, determines completely the way values of that abstract syntax are to be encoded/decoded.

Who cares about encodings, provided they work? Well, some people want to take control!

1.2 Can ECN be used alone?

No. ECN is always used to specify the encoding of one or more ASN.1 types. We normally refer to an "ASN.1+ECN" specification.

ASN.1+ECN is the phrase to use.

1.3 What are its applications?

1.3.1 Specialized encodings of simple fields

ECN specifications can be used to provide "specialized encodings" for some parts of an ASN.1 specification.

PER is mostly OK, but a few critical fields need to be done differently? OK. Use ECN.

For example, ECN can be used to specify that identified INTEGER fields in an ASN.1 specification are to be encoded using Huffman encoding techniques.

A Huffman encoding uses the theoretical-minimum number of bits (averaged over many transmissions) to transfer integer values, given the probabilities associated with the occurrence of each possible value of the INTEGER field.

The same mechanism can be used to define optimum "identifier" values for messages, given the frequency of their occurrence, and optimum encoding of "choice determinants" to identify the selection of alternatives in an ASN.1 choice construct.

1.3.2 Specialized encodings of constructors

Frequently a designer will wish to use alternative mechanisms to those of standard encoding rules for some constructs.

I don't want a length count - I want to terminate my character strings with a null octet!

For example, a "SEQUENCE OF A" can be supported by using a "More-bit" set to 1 before each instance of "A", with a terminating "More-bit" set to zero at the end of the sequence-of. (In the case of zero iterations, there will be a single zero bit.)

ECN can be used to specify the use of this mechanism for some of the sequence-of constructions in a specification, whilst retaining the standard encoding (a count of iterations) for other instances of sequence-of.

Another feature of ASN.1 that is described as a "constructor" (as opposed to a bit-field in the encoding) is an occurrence of OPTIONAL or DEFAULT. A "Presence-bit" set to one if the optional element is present, to zero otherwise, is a common form of encoding, but PER groups all such present bits at the head of a sequence. ECN can be used to place all presence bits immediately before the element whose presence or absence they determine.

It should be noted that there are some forms of "optionality" that are not easily and cleanly supported in ASN.1 without application rules. One of these is the presence of a sequence of optional elements where it is a requirement that an optional element can be present only if all the preceding optional elements of the sequence are present. Typically, in this case, there is no "Presence-bit", rather the last optional element is determined by a receiver when the end of a PDU, or of a length-delimited sequence, is encountered. This mechanism is sometimes called use of "tail-end optionals".

Whilst these application rules cannot be formally expressed in ASN.1, ECN can be used to generate the required encodings.

1.3.3 Specialized mechanisms for extensibility

Everybody has their own approach to extending protocols.

"Extensibility" is an ASN.1 term, which refers to the totality of specifications on the behavior of Version 1 systems that enables Version 2+ systems to interwork with them in a simple manner, without the need for Version 2+ systems to use dual-stack approaches. (A dual-stack approach is one where a significant part of the code for earlier versions has to be included in parallel with the code for later versions, rather than being able to use – part of – the code of the later version to interact with later versions.)

Extensibility generally involves the specification of places in an encoding where Version 2+ material will be added, with mechanisms for Version 1 systems to detect the end of such encodings, and to ignore them. Such encodings may occur at the end of PDUs or other length-delimited constructs, or may appear at identified places within an encoding.

Extensibility is signaled in an ASN.1 specification by the use of the extension marker (and version brackets). The effect such notation has on encodings, and the decoding rules for Version 1+ systems, can be specified using ECN.

The "tail-end optionals" mechanism described above can be extended very simply to provide support for extensibility, and its use for this purpose can be specified using ECN.

1.3.4 Re-definition of legacy protocols

How ECN can take over the world!

Defining specialized encodings for some ASN.1 fields or constructors (or for extension mechanisms) may occur in newly-defined ASN.1 protocols where the encoding is required to be even more efficient than a standard PER encoding. However, the most common use for such specialized encodings is to support a "legacy protocol".

In this scenario, we have a long-established protocol defined by some form of ad hoc notation - for example pictures of bits and bytes, or some form of tabular notation.

Encode/decode routines for such protocols cannot be provided by application-independent tools. Such protocols frequently (have to) complicate the specification for implementors of the application by making explicit and ad hoc reference to (and specification of) "Presence-bits", "More-bits", length fields, and fields which determine which of a number of possible alternatives has been encoded (choice determinants).

There have in the past been many attempts to express such protocols using ASN.1, but this has served only the purpose of clarifying the structure (abstract syntax) of the application data. Standard ASN.1 tools applied to such structures would in general not generate the same bits-on-the-line as the original specification.

Attempts to write the ASN.1 in such a way that the bits on the line **were** the same was sometimes just about possible, but only by including in the ASN.1 as explicit components of the abstract syntax the "auxiliary fields" used to support the encoding - the presence bits, length fields, etc.

With ECN, it is possible to write "clean" ASN.1, containing only components that carry application semantics, and to generate from that an encoding structure that contains all the auxiliary fields. ECN further permits the definition of encodings for that generated structure which will produce exactly the bits-on-the-line that are required by the original specification.

It may be that few authoritative, standardized, definitions of legacy protocols will be re-defined using ASN.1+ECN. However, the re-definition of the protocol in this way by **implementors** may become quite common, in order to gain access to ASN.1+ECN tools to perform the encode/decode functions of the application.

A later version will mention the advantages provided for testing purpose (a link with TTCN) and advantages for formalization of the protocol behavior in SDL (a link between SDL, TTCN, ASN.1 and ECN could be provided within SG10).

1.4 Is ECN powerful enough for all these purposes?

Specification of ECN has been a learning process. The requirements of several groups for specialized encodings, but particularly the sorts of encodings used in legacy protocols, have been carefully studied.

Well, it seems to do most things, and it can always be added to!

It seems likely that the ECN framework has sufficient functions and flexibility to satisfy all the needs related to applications of ECN identified above.

In detail, however, additions to ECN functionality may occur of the next few years as further encoding mechanisms are uncovered and support for them is added.

Potential ECN users are encouraged to ask for additional functions if the present specification fails to meet their needs.

1.5 Is ECN simple and clear enough for use and implementation?

Once the fundamental principles of ECN are understood, it is not too difficult to learn to write it, but a good knowledge of ASN.1 makes the task a lot easier.

Simplicity is in the eye of the beholder!

The notation is (designedly) quite similar to the ASN.1 notation, and many ASN.1 concepts carry across. Thus parameterization is available in ECN definitions in exactly the same way as in ASN.1. The relationship of values to types, information objects to information object classes, carries across to the relationship of ECN **encoding objects** to **encoding classes**. The notational structures used for defining ECN encoding objects of some encoding class and for defining encoding object sets will be familiar to those familiar with ASN.1 notation for defining values and information object sets or value sets. The notation used to define an ECN **encoding structure** is very similar to the definition of ASN.1 types (with curly brackets and commas used in a similar way). However, the "#" character is used at the start of names and provides immediate recognition that what is being defined is an encoding structure and not an ASN.1 type.

Tools are available to syntax check ECN specifications. Tools which will enable implementation of a protocol to be produced as easily from an ASN.1+ECN specification as from an ASN.1 specification that uses standardized encoding/decoding rules have been announced. (Contrast ASN.1, where even syntax checkers did not become available until several years after the first Standard/Recommendation was produced.)

As with any notation (including ASN.1!) it is possible to write obscure and complex ASN.1+ECN, but provided the appropriate mechanisms are used, ECN specifications can be very straight-forward and obvious once the basic concepts have been grasped and the syntactic framework understood.

2 History and current status of the ECN work

You either find a historical perspective illuminating or totally boring – skip this clause if you wish!

This text was written December 2000, and reflects not only the document under ballot, but also ballot comments that, if accepted, will add additional functionality. It is expected that the output of the January 2001 Editing Meeting will contain all the features described in this tutorial, but readers are cautioned that there could still be changes in detail, and even further additions of functionality.

Version 2 of this Tutorial is expected to be issued shortly after the January Editing Meeting, and is likely to be more "stable" than this version 1. As stated at the head of the document, an expiry date for this text of Spring 2001 is probably about right.

The idea of defining appropriate encoding rules to enable ASN.1 to be applied to legacy protocols (with the same bits on the line) originated with several workers during the 1980s.

The approach was to define Minimum Bit Encoding Rules (MBER) which would encode integer values into the minimum bits required by its constraints. If an ASN.1 type was defined with a component for every field of the encoding (suitably constrained), and MBER was then applied, the legacy encoding would be obtained.

This approach never worked except in very trivial cases. One reason was that **auxiliary fields** are frequently present in an encoding (length determinants, presence bits, etc). It was possible to generate encodings for these by including components in the ASN.1 specification for them, but there was no way to express the fact that these fields controlled the length or presence of other fields. Purists also regarded it (and still regard it) as unsatisfactory to include such auxiliary fields (concerned solely with encoding mechanisms) in the abstract syntax definition.

However, when the ASN.1 Packed Encoding Rules were developed, many of the MBER concepts were incorporated. As a result, to this day, unaligned PER is generally used as the base encoding (modified where necessary by use of ECN) when legacy protocols are to be defined by ASN.1+ECN.

The idea of an Encoding Control Notation separate from, but applied to, an ASN.1 specification was first conceived by Frank Schramm of Siemens and by Colin Wilcock of Nokia. Their early work also recognized that there were two types of fields in an encoding:

- Fields carrying application semantics; and
- Auxiliary fields used only to support the encoding.

Like early workers, however, their proposals were based on including auxiliary fields in the ASN.1 specification (the so-called "one-to-one" model), but they did start to address the problem of identifying some fields as a presence or length determinant for other fields. Today, the determinant concept remains a primary part of ECN, but it is now considered a bad design to have ASN.1 components corresponding to auxiliary fields - such fields can and should be inserted as necessary into encoding structures generated from the ASN.1 type definitions. ECN provides several mechanisms for doing this.

These two workers first proposed that a Recommendation/Standard be developed for Encoding Control (EC) - later to become Encoding Control Notation (ECN) - in January 1999. It was, however, about twelve months later before the ideas of ECN began to be established, and it was not until a major ETSI input - a complete new text for the Recommendation/Standard - was produced in the summer of 2000 that ECN in its present form began to mature.

The early work still relied on putting auxiliary fields into the ASN.1 specification - something that purists strongly objected to. It was not until much later in the work that the concept emerged of supplementing the ASN.1 specification with an **encoding structure definition** that would unashamedly contain a field for every field in the encoding. Strong links were, of course needed between the notation used to define encoding structures and ASN.1 itself.

Today encoding structures are typically produced by automatic generation from an ASN.1 type definition, with ECN notation selectively changing parts of the structure, replacing parts, adding fields, and so on, before the application of actual encodings to the structure.

A crucial development in the ECN work (in late 1999) was the introduction of the concepts of encoding objects and encoding classes, mentioned earlier. Another key mechanism was the introduction (in Autumn 2000) of "coloring" a structure generated from an ASN.1 type to identify which constructors in it require specialized encodings. Throughout the work, means of **easily** specifying replacements of parts of a structure with more complex structures was a continuing development. One important mechanism was to allow an encoding object for a constructor to perform replacement of the entire construct or of components before applying encodings. These concepts are described later in this tutorial.

Following the ETSI input in the Summer of 2000, the ECN work matured rapidly, and began a six month Final Committee Draft (FCD) ISO/IEC ballot. The FCD ballot closes late January 2001, and the expected final approval of the first ECN Standard/Recommendation is expected around Summer 2001.

3 Fundamental concepts

This section contains some examples to illustrate the description and to familiarize the reader with the ECN syntax. Additional small examples can be found in an Annex of the ECN Standard itself. Finally, a number of "ETSI ECN Case Studies" of the application of ECN to legacy protocols are in preparation, and will form a further source of large and real-world examples.

3.1 Structure of an ASN.1+ECN specification

An example of the structure of ASN.1 and ECN definitions appears at the end of this sub-clause.

Recognize the top-level constructs, and you will begin to feel comfortable!

3.1.1 Structure concepts

Readers will be familiar with the concept that an application is defined using one or more ASN.1 modules, with import and export of types, values, information object classes, information objects, and information object sets between these modules.

In an ASN.1+ECN specification, these ASN.1 modules are supplemented by a set of ECN modules, with export and import between both the ECN modules and between ECN modules and ASN.1 modules.

There are two sorts of ECN modules: Encoding Definition Modules (EDM) and an Encoding Link Module (ELM). Although the expansion does not read well ("Module module"), it is common to refer to EDM modules and to ELM modules.

For any given application whose messages are defined using ASN.1+ECN, there will in general be any number of ASN.1 and EDM modules, but precisely one ELM module.

The term "Encoding Link Module" was coined when the sole purpose of this module was to link together the ASN.1 and ECN specifications, and the name has been retained. Today, its main function is to import structures corresponding to ASN.1 types, encodings from EDM modules, and to define the encoding of the ASN.1 types by applying the encodings to the corresponding structures. (Thus the name "Encoding Application Module" might be more appropriate, but is not used.)

This separation of EDM functionality (defining a set of possible encodings of bit-fields and of constructors) from ELM functionality (applying the encodings to determine the encoding of ASN.1 types) is fundamental to an understanding of ECN. It is only in the ELM that one finds the authoritative definition of how any type in an ASN.1 module is to be encoded.

In order to know the encoding of any top-level application message defined using ASN.1, the first requirement is to locate the ELM. Justifying its name, this will link to ASN.1 modules containing the type definition for the message, and to EDM modules containing the definition of the encodings that it applies to those types.

3.1.2 Structure examples

An ASN.1+ECN specification consists of one or more ASN.1 modules (totally standard ASN.1, no additions, no restrictions):

```
ASN1-module-my-name
{joint-iso-itu-t(2) To be supplied}
DEFINITIONS ::=
BEGIN
EXPORTS ......;
IMPORTS ......;
.....
END
```

together with one or more EDM modules:

```
EDM-module-my-name
{joint-iso-itu-t(2) To be supplied}
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS ......;
GENERATES-AND-EXPORTS ......;
IMPORTS ......;
......
END
```

The "GENERATES-AND-EXPORTS" clause references an ASN.1 module for a structure corresponding to an ASN.1 type definition, and generates (and exports) a new structure with some replacements and additions. This is discussed more fully under "coloring" below.

The specification is completed with a single ELM module:

```
ELM-module-my-name
{joint-iso-itu-t(2) To be supplied}
LINK-DEFINITIONS ::=
BEGIN
IMPORTS ......;
......
END
```

Within the body of EDM modules, assignments used to define encoding objects, encoding object classes, and encoding object sets take the familiar ASN.1 form using "::=". In the ELM the statements are applying encodings, and use a slightly different syntax discussed later.

3.2 Names of encoding structures

What's in a name? Scope rules? Yuck!

As for ASN.1 types, different modules may define encoding classes with the same name. If types with the same name are exported from different ASN.1 modules and then imported into a single ASN.1 module, there is potential for confusion. ASN.1 resolves this by the use of an **external reference** to the type. An absolute reference is what we call here a "fully-qualified name" which includes the name of the module that defined the type.

In ECN, these problems become more acute, as things called "explicitly generated structures" have the same simple class name as things called "implicitly generated structures", but are defined in different modules. The use of a fully-qualified name to distinguish such structures is important in the use of ECN.

3.3 Application fields and auxiliary fields

Determination is what is required!

This is a short sub-clause. The concepts have already been mentioned, and there is no associated syntax to examine.

Consider an ASN.1 type definition, and an encoding produced by the application of standard encoding rules.

The encoding consists of a number of identifiable fields. Some of these carry application semantics and correspond directly to components of the ASN.1 type definition. These are called **application fields** in the encoding. Others are inserted by the encoding rules to enable decoding to take place.

There are many names given to these auxiliary fields by different authors, but in ECN work the term **determinant** is usually employed. So a bit that determines whether an optional element is present or not is called a **presence determinant**. A field that determines the length of an element is called a **length determinant**, and a field that determines which of two possible alternatives is present is called a **choice determinant**.

For PER encodings, these fields can be clearly identified, and perform a single function, but the precise way they work can vary. Thus a length determinant in PER may be a count of bits, of octets, or of iterations. In PER - but in few other encoding schemes - the presence determinants for all the optionals in a sequence are collected together at the head of the sequence. In BER, there are length determinants, even in cases where they are not strictly needed for decoding, and the encoding of the tag performs the roles of presence determinant and of choice determinant.

A later part of this tutorial discusses some of the many mechanisms that can be used for determination. ECN has to support the insertion of auxiliary fields in a flexible manner, and has to support the many ways in which such fields are used for determination.

3.4 Encoding Classes, Encoding Objects, and Encoding Object Sets

Understand this stuff, and all else is easy! Well .... fairly easy!

This is the most important fundamental concept in the ECN work, and ECN cannot be understood or used without a good grasp of the concept. The concept is introduced at this point in the tutorial, but in a fairly vague way, and with reference to ASN.1 type constructs. We will see later that it actually applies to encoding structure fields, and not to parts of ASN.1 type definitions, and we will later provide a more precise

understanding of the idea of "synonymous" classes. The introduction to the concepts in this clause does not tell any lies, but may leave the intelligent reader asking some further questions.

The basic idea is that we can produce a definition of how to encode a boolean. For the moment, let us keep it simple - we can either:

- Use a single zero bit to represent TRUE and a single one bit to represent FALSE. Or
- Use a full octet of zeros to represent TRUE and any non-zero value to represent FALSE (or vice versa).

Of course, there are many other possible encodings for a boolean. Each of these definitions of how to encode a boolean is called an **encoding object** for the #BOOLEAN **encoding class**. (Note the "#" - all encoding class names start with a "#".

Clearly any of the specifications of how to encode a boolean can be used to encode a boolean field in a protocol, but they would not be of much use in trying to encode an integer field, or a sequence-of constructor.

Thus we talk about having different encoding classes #BOOLEAN, #INTEGER, #SEQUENCE-OF, #CHOICE, and so on. In order to specify encodings for a complete protocol, we generally need a set of encoding objects, one for each class that is present in the type definition. (To be more precise, one for each class that is present in the encoding structure that corresponds to the type definition.)

As noted above, names of encoding classes always start with a "#" followed by an upper-case letter. This is done to help human readers recognize when an encoding class name is being used. It is not strictly needed for computers.

Some encoding classes are regarded as synonyms for other encoding classes, because the functionality needed to define encoding objects for those classes are so similar, and the same syntax is used for their definition. Thus the #ENUMERATED class and the #INTEGER class are defined as synonyms. Similarly, the #SEQUENCE-OF and #SET-OF classes are also defined to be synonyms.

The main importance of synonyms is simply that the same syntax is used for defining encoding objects of synonymous classes. However, we will see later that the ability to define is important if some (e.g.) SEQUENCE OF constructions in an ASN.1 message are to have their lengths determined one way, and some a different way.

If an encoding object is defined for a particular class, it normally encodes precisely that class, not a synonymous class. (See, however, the discussion on the application of encodings later in this Tutorial.) Thus we model PER encodings by saying that PER defines encoding objects for the #SEQUENCE class and for the #SET class (the former giving a textual ordering of components and the latter a tag-based encoding). When PER encodings are applied, the encoding object for the #SEQUENCE class "latches on" or "attaches itself" to all the encoding structure fields corresponding to the ASN.1 constructor SEQUENCE, whilst the encoding object for the #SET class "latches on" or "attaches itself" to fields corresponding to occurrences of SET.

When performing specialized encodings it can often happen that some instances in a protocol of ASN.1 sequence-of constructs are to be encoded in one way, whilst others are to be encoded in a different way. To support this, occurrences of (for example) sequence-of can be "colored" so that some map into one class and others map into a different (synonymous) class, enabling different encodings to be applied to the different occurrences.

There is a strong analogy between the ASN.1 "value of a type" concept and the ECN "encoding object of a class" concept. Thus:

my-int INTEGER ::= .....

in an ASN.1 module defines a value ("my-int") of type INTEGER, with syntax on the right of the "::=" being dependent on the type of the "governor" ("INTEGER" in this case). Similarly:

my-boolean-encoding #BOOLEAN ::= .....

in an EDM module defines an encoding object ("my-boolean-encoding") of encoding class #BOOLEAN, with syntax on the right of the "::=" being dependent on the class of the "governor" ("#BOOLEAN" in this case).

3.5 Encoding Structures

3.5.1 Introduction to the concept

As ASN.1 is to abstract values, so ECN is to fields of encodings: combine them, choose between them, miss them out, repeat them, it is all just the same construction mechanisms.

Encoding structure definitions describe the structure of an encoding. As with abstract values, simple encoding fields are grouped into composite units by encoding constructors. Such constructions can be named, and can then be used within the definition of more complex encoding structures. The concepts and syntax are similar to the ASN.1 concepts of type definition, but are here referring to fields in encodings, and to the rules for their combination.

The ECN model enhances the meaning of an ASN.1 module type definition (but does not in any way change the syntax). When an ASN.1 type is defined, an **implicitly generated encoding structure** is also defined, and automatically exported from that ASN.1 module.

This encoding structure contains fields (and encoding constructors) corresponding to the components and constructors used in the ASN.1 type definition, and its exact form is fully specified in the ECN Standard.

Very roughly, a text representation of that implicitly generated structure would look like the ASN.1 type definition, but with a # before the names. (There are a number of other differences, however, described below.) The name of an encoding structure corresponding to an ASN.1 type is always the ASN.1 type name preceded by a "#". For example, defining the ASN.1 type "My-Type" in an ASN.1 module would implicitly generate (define and export) a corresponding encoding structure called "#My-Type".

It is important to realize, however, that despite looking a bit like a type definition, an encoding structure definition is very different. Here we are dealing with fields in an encoding (concrete syntax) and the way that they are combined, **not** with abstract values (abstract syntax) and the way abstract values are combined.

As well as being implicitly generated by ASN.1 type definitions, encoding structures can be produced and named using ECN syntax. This is an important part of ECN, as it enables parts of an implicitly generated encoding structure to be replaced by another (usually more complex) encoding structure.

When such replacement occurs, there is a mapping of abstract values ("value mappings") from the original structure to the replacement structure. Encodings of the replacement structure (for these abstract values) now form the encodings of the abstract values present in that part of the original structure that has been replaced. "Value mappings" are discussed further later.

Returning to the meaning of "encoding class": an encoding class is a part of an encoding structure definition, and represents the set of all possible encoding specifications (encoding objects) related to that part of the encoding structure. Encoding classes may be concerned with the production of actual bits in the encoding, or with procedures which determine how fields are concatenated, alternatives identified, repetitions delimited, and so on.

3.5.2 Examples of encoding structure definitions

Lot's of simple concrete examples - all of them littered with trash (sorry, HASH!)

The time has come to present encoding structure definitions that use most of the available syntax. This syntax is presented in stages, with the definition of "#Structure1", "#Structure2", etc. The following text comments on these structure definitions. Structure definitions appear within (and are imported and exported between) EDM modules.

```
#Structure1 ::=  #CONCATENATION {
     field1  #INT,
     -- Comment can be embedded, as in ASN.1
     field2  #BOOL,
     field3  #NUL,
     field4  #PAD,
     field5  #ALTERNATIVES {
           alt1  #CHARS,
           alt2  #BITS },
     field6  #REPETITION {
          #OCTETS },
     field7  #Structure2 }
```

This definition uses the main primitive classes that would normally be used in an explicit definition of an encoding structure.

Here only primitive fields are used (except for "#Structure2"), but we can alternatively define reference names for some parts of the definition and then use those reference names, just as in ASN.1. This is illustrated by the use of "#Structure2" (defined below).

We see here three constructor classes: #CONCATENATION, producing a field that is a concatenation of other fields (we will see later that some of these fields can be marked optional); #ALTERNATIVES, producing a field that is one of several alternative fields; and #REPETITION, producing a field that is a series of repetitions of a given field. (These play a similar role in encoding structure definition as do SEQUENCE (or SET) and CHOICE and SEQUENCE OF (or SET OF) in type definition).

Note that the syntax for #REPETITION is a pair of curly braces enclosing the component, a change from similar ASN.1 syntax, and designed to make the ECN syntax easier for computers!

Encoding objects of these constructor classes specify how their component fields are composed to form the more complex field. This includes termination mechanisms for a #REPETITION, and resolution of which alternative is present for an #ALTERNATIVES.

In addition to the constructor classes, we have a number of bit-field classes used - classes whose encoding objects actually produce bits in the encoding. These bit-field classes can have associated with them abstract values, and encoding objects for these classes are required to produce distinct encodings for each of the abstract values assigned to the class.

The following table gives the type of abstract value that can be associated with each class. The name of the class differs slightly from the name of the ASN.1 type that is associated with such values, in order to emphasize that here we are concerned with the set of all possible encodings for those abstract values, not simply with a set of abstract values. The table is:

#INT	**Integer values**
#BOOL	**Boolean values**
#NUL	**The null abstract value**
#PAD	**No abstract values can be associated with this class. It never carries application semantics.**
#CHARS	**Character string values**
#BITS	**Bit string values**
#OCTETS	**Octet string values**

The association of abstract values (part of some ASN.1 type) with one of these fields is done either implicitly when a structure is generated from an ASN.1 type, or by the use of value mapping.

We define "Structure2" as:

```
#Structure2 ::= #CONCATENATION {
    field1  #OBJECT-IDENTIFIER,
    field2  #RELATIVE-OID,
    field3  #OPEN-TYPE,
    field5  #REAL,
    field6  #Structure3 }
```

This structure contains primitive classes that can carry object identifier, relative object identifier, open type, and real values. These classes are currently "second-class citizens". There are no mechanisms for directly defining encodings for these classes (other than use of a "user-defined function" - see later).

The problem is partly that the abstract values associated with these classes are fairly complex abstract values, and the functionality needed to determine encodings for such values in a general way has not yet been determined.

We define "#Structure3" as:

```
#Structure3 ::= #TAG #CONCATENATION {
    field1  #BOOL,
    field2  #INT #OPTIONAL,
    field3  #TAG #Structure4,
    field4  #INT (0..7),
    field5  #OCTETS (SIZE (6)) }
```

Encoding objects of the #TAG class determine the encoding of the numerical value (the tag class is ignored) of the outermost tag of the abstract values associated with the following class. (#TAG classes are normally only included in implicitly generated encoding structures, but can also be included in explicitly-defined structures.)

The presence of an #OPTIONAL class is permitted only within a #CONCATENATION (or within a synonym of that class - see later), and identifies that the corresponding bit-field class encoding may be absent. Encoding objects of this class specify how the total encoding is produced in order to enable a decoder to determine whether the element is present or not. This often - but not always - implies the use of a #BOOL field (perhaps "field1") within the sequence, acting as a presence determinant. Explicit presence determinants are only one of several ways of resolving the presence of an optional element.

The "(0..7)" and "(6)" notations are similar to bounds notations in ASN.1. The first indicates that only abstract values in the range zero to seven can be encoded by the field (and hence restricts the abstract values that can be mapped to that field). The latter indicates that the field is exactly six octets long.

We define "#Structure4" as:

```
#Structure4 ::= #CONCATENATION {
    field1 #INT,
    #EXTENSIONS {
        #VERSION-BRACKET {
            field1   #INT,
            field2   #BOOL } ,
        #VERSION-BRACKET {
            field1   #CHARS,
            field2   #OCTETS } } ,
    field2 #BOOL,
    field3 #Structure5 }
```

Support for extensibility may be delayed for an amendment to the first ECN Standard/Recommendation, as work in this area is not fully mature, but the above illustrates the likely form that such support will take in an encoding structure definition.

ASN.1 notation only supports a single insertion point in a SEQUENCE (or SET). Multiple insertion points can, however, be specified by using SEQUENCE constructions in tandem, and it is likely that ECN will (unnecessarily) restrict use of #EXTENSIONS to a single occurrence within any #CONCATENATION.

Encoding objects of the encoding class "#VERSION-BRACKET" determine any wrapper to be placed about the components (which in ASN.1 may not contain nested extensions - it is not yet determined whether ECN will impose the same restriction). There are many actions that encoding objects of the #VERSION-BRACKET class (and of the #EXTENSIONS class) may specify, but the available actions are broadly similar to those for #SEQUENCE, but with an in-built differential encode/decode action (see later).

Similarly, encoding objects of the encoding class #EXTENSIONS defines wrappers and differential encode/decode for its "components" - the #VERSION-BRACKET fields.

We define "#Structure5" as:

```
#Structure5 ::=  #SEQUENCE {
    field1  #INTEGER,
    -- Looks rather like #Structure1, yes?
    field2  #BOOLEAN,
    field3  #NULL,
    field5  #CHOICE {
          alt1  #UTF8String,
          alt2  #BIT-STRING },
    field6  #SEQUENCE-OF {
          #OCTET-STRING } }
```

This looks much more like ASN.1! In fact, it is using built-in synonyms defined as:

```
#SEQUENCE          ::= #CONCATENATION
#INTEGER           ::= #INT
#BOOLEAN           ::= #BOOL
#NULL              ::= #NUL
#CHOICE            ::= #ALTERNATIVES
#UTF8String        ::= #CHARS
```

```
#BIT-STRING          ::= #BITS
#SEQUENCE-OF         ::= #REPETITION
#OCTET-STRING        ::= #OCTETS
```

There are built-in synonyms for names corresponding closely to the ASN.1 built-in types. Why are synonyms important? And what about assignments like:

```
                #SEQUENCE-OF        ::= #REPETITION
or              #My-sequence-of     ::= #SEQUENCE-OF
```

We are used in ASN.1 to define synonyms for types such as INTEGER, BOOLEAN, etc (typically the synonyms signal different semantics).

This is also possible in ECN for bit-field classes, but ECN extends this to constructor classes such as #REPETITION. Why?

We will see later that in applying encodings to a structure, we use a set of encoding objects, **with no more than one object for any given class in the set**.

If we are looking for an encoding for a "#My-sequence-of" constructor, and there **is** one in the encoding object set being applied to the structure, then that encoding object determines the encoding for that field of the encoding structure.

If there is no encoding object of the class "#My-sequence-of", then the reference is resolved, and we try to find one of the #SEQUENCE-OF class. If there isn't one of those, then we resolve the reference again and try for an encoding object for #REPETITION.

Why does this matter? ECN is very much about providing specialized encodings for some (but not all) concatenations, repetitions, and alternatives in a structure derived from an ASN.1 type. It is important, therefore, to be able to identify which concatenations are encoded in a specialized way, and which in a standardized way. (There may, of course be several "specialized ways" used in different parts of the ASN.1 definition.)

The synonym mechanism, and the rules for applying encodings (see below) cover this requirement.

3.6 Categories of classes

If I define "#My-class" as a synonym for #OPTIONAL, can I use it in a structure definition where (for example) a #CONCATENATION might be expected? Of course not!

Categories of categories of categories - we love to categorize.

ECN identifies six different categories of classes. If a new class name is given to an existing class, the category of the old class becomes the category of the new class, and it can only be used in structure definition where that category of class is allowed.

The following categories of class are defined:

Bit-field classes	**Things derived from #INT, #BOOL, etc**
Concatenation classes	**Things derived from #CONCATENATION (for example, #SEQUENCE, #SET, #My-sequence)**
Optional classes	**Things derived from #OPTIONAL (for example, #Trailing-optionals)**
Identification classes	**Things derived from #TAG (for example, #Tag-in-6-bits)**
Repetition classes	**Things derived from #REPETITION (for example, #SEQUENCE-OF, #SET-OF, #More-bit-sequence-of)**
Alternatives classes	**Things derived from #ALTERNATIVES (for example, #CHOICE, #ID-selected-choice)**

An encoding structure can contain any of these classes (primitive, built-in, or user-defined), provided the category of the class is the right one for its use at that position in an encoding structure definition.

As an aside: Allowing user-defined synonyms for constructors like #CONCATENATION produced ambiguity in the structure definition syntax when parameterization (discussed later) was used. It was for this reason that parameter lists in ECN begin with "{<" and end with ">}", rather than the simple "{" and "}" that ASN.1 uses.

3.7 Implicitly generated encoding structures

All the above syntax can be used in two different circumstances:

Don't worry - writing structures is all done for you! Just get your ASN.1 right!

- The explicit definition of encoding structures used in the definition of the encoding of a complete ASN.1 type (by replacement and value mapping - see later).
- The definition of (replacement) structures for components of a structure (perhaps including the insertion of "head-end" structures). This is needed to support the specification of encodings for constructors such as #CONCATENATION and its synonyms.

The use of encoding structures does, however, go far beyond the explicit definition of encoding structures by an ECN definer.

As has been indicated earlier, all ASN.1 type definitions implicitly define (and export from their module) a corresponding encoding structure.

This **implicitly generated encoding structure** has fields, which have associated with them abstract values of components of the ASN.1 type. The end result is that encodings of the abstract values of the complete ASN.1 type can be defined by encodings of the implicitly generated structure.

It is this structure (of encoding classes) that is encoded (by a set of encoding objects) to determine the encoding of values of the ASN.1 type.

The ECN specification completely defines (algorithmically) the implicitly generated encoding structure for any type that can be defined using ASN.1. The implicitly generated structure uses built-in encoding class synonyms, so that the names of classes are similar to the ASN.1 built-in type and constructor names. This also ensures that information is not lost on the distinction between "ENUMERATED" and "INTEGER", or between "SEQUENCE" and "SET", etc. These ASN.1 notations represent the same primitive classes, but the encodings applied to them are typically different (and are certainly different in the standardized ASN.1 encoding rules).

The mapping between ASN.1 type definitions and encoding structures is "obvious", given the names of the synonyms for the classes. The reader of Version 1 of this Tutorial is referred (sorry!) to the ECN Standard for details of the correspondence, but a few points can usefully be made.

In general, "frills and fancies" in the ASN.1 definitions are resolved and eliminated before the corresponding encoding structure is generated.

For example:

- All value references are completely resolved.
- All non-PER-visible constraints are discarded - yes, ECN links to PER for the definition of constraints that can be used in determining encodings. If a constraint does not affect PER encodings, then it cannot be used to change the encodings defined by ECN. It is highly unlikely that this will give rise to limitations in the practical use of ECN.
- All integer and length constraints are resolved to a simple pair of bounds (which might use "MIN" and "MAX", but are otherwise self-contained), and alphabet constraints are resolved to the effective alphabet constraint.

  NOTE — These constraints can be referenced in the definition of encoding objects, so a #INT with all-positive values can be encoded differently from one with both positive and negative values, and a #CHARS can encode differently if it has different effective alphabet constraints.
- Named bits in BIT STRING specifications, and distinguished values in INTEGER specifications are not visible to ECN.
- Names of enumerations are lost, and any automatic allocation of values to enumerations are applied - the ENUMERATED type maps into the #ENUMERATED synonym of #INT, with appropriate bounds.
- All parameterization is fully resolved, with parameterised types and values ignored and instantiations of those replaced by the corresponding parameterised type with the actual parameters inserted.

- All occurrences of "COMPONENTS OF" and of "SelectionType" are expanded.
- Both OPTIONAL and DEFAULT ... map into #OPTIONAL.
- Various syntactic constructs related to information object definition are resolved to their underlying ASN.1 types.
- etc etc.

There are special rules concerned with the presence of a contents specification (introduced by “CONTAINING” and “ENCODED BY”) on an OCTET STRING or BIT STRING. These go beyond the scope of this Tutorial.

The user of ECN can sometimes ignore concerns with the details of the implicitly generated encoding structure, but for detailed control of encodings, a familiarity with this structure is important, and the reader is referred to the ECN Standard.

There are three rather important points to make/re-iterate to conclude this sub-clause.

First, there is an implicitly generated (and automatically exported) encoding structure for each ASN.1 type definition, with a name that is the same as the ASN.1 type definition, preceded by a "#", and which has associated with it all the abstract values of the ASN.1 type. ASN.1 type names such as "INT", which are allowed in ASN.1, but where the corresponding class name (e.g. "#INT") is a reserved name in ECN, cause some difficulty, but these cases are covered by the ECN Standard. Essentially, if such classes are referenced in an ECN specification, then the class name has to be qualified by the name of the module in which it is implicitly generated, using the "external reference" form of class name (similar to "external references" for type names in ASN.1).

Second, the implicitly generated structure may be transformed by ECN specifications into other structures, usually by replacement of some of the classes within it by other classes (which are either synonyms for constructors, or are more complex bit-field structures). Such transformations include defined mappings of the abstract values from (parts of) the structure being replaced to (parts of) the replacement structure.

Third, the actual encoding of an ASN.1 type is totally determined by the application of encodings to the implicitly generated structure, or to a structure which is derived from it (and to which its abstract values have been mapped).

Here are some simple ASN.1 type definitions:

```
Color ::= [1] INTEGER {red(10), orange(20), yellow(30),
                       green(40), blue(50), indigo(60), violet(70)}

Version ::= [2] BIT STRING {version1(0), version2(2)}

My-type ::= SEQUENCE {
  field1     Version,
  field2     Color,
  field3     CHOICE {
     alt1    [3] INTEGER (0..15),
     alt2    [4] INTEGER (16..MAX) },
  field4     SEQUENCE OF
             OCTET STRING (SIZE (0..10)) DEFAULT ''B}
```

Here are the implicitly generated structures:

```
#Color ::= #TAG #INTEGER

#Version ::= #TAG #BIT-STRING

#My-type ::= #SEQUENCE {
   field1     #Version,
   field2     #Color,
   field3     #CHOICE {
      alt1         #TAG #INTEGER (0..15),
      alt2         #TAG #INTEGER (16..MAX) },
   field4     #SEQUENCE-OF {
                 #OCTET-STRING (SIZE (0..10)) #OPTIONAL }
```

3.8 Applying encodings

A brief re-cap:

This is what it is all about. How to encode your top-level messages.

- Every ASN.1 type definition has a corresponding encoding structure (either implicitly generated, or derived from the implicitly generated structure).
- Specification of an encoding for an ASN.1 type consists of applying (in the ELM) a set of encoding objects to the corresponding encoding structure.
- Each encoding object in the encoding object set being applied is for a distinct encoding class. (An encoding object set is **never** allowed to contain multiple encoding objects for the same encoding class.

When an encoding object set is applied to an ASN.1 type (defined within or imported into a specified ASN.1 module), it applies to all the parts of that type (with controlled resolution of all type references). However, it does in no way imply the encoding of that type when used as a type reference within some other containing type. (This tutorial ignores the case of a type that is recursively defined. This produces definitional problems, but no real problems of intent.)

The process of applying an encoding object set to a structure (which corresponds to an ASN.1 type, and whose encoding defines the encoding of that type) proceeds as follows:

- If there is an encoding object in the set for the class of the entire structure (for example, "#My-Messages"), then that is applied, and it completely determines the encoding of the type "My-Messages". (This case would be unusual).
- Otherwise, the class name is de-referenced (which usually means in practice that the type-reference name is de-referenced).

De-referencing will normally produce a structure such as:

```
#My-Sequence-Class {
    field1  #Type1,
    field2  #Type2,
    ......
    fieldn  #Typen }
```

Where #My-Sequence-Class is a synonym for #CONCATENATION.

At this stage, we look to see if the encoding object set contains an encoding object for "#My-Sequence-Class". If found, then this encoding object is applied, and the whole original encoding object set is then applied (in parallel, and independently) to each component of the structure.

Doing this recursively produces the final encoding of all components (to any depth) of the original type.

If there was no encoding object for the constructor #My-Sequence-Class, then this is de-referenced to produce (say) #SEQUENCE, which can if necessary be further de-referenced (using the ECN built-in assignment "#SEQUENCE ::= #CONCATENATION") to produce #CONCATENATION. If there are no encoding objects of the class #SEQUENCE or #CONCATENATION in the set, then the specification is in error.

Note that the encoding object set can (and is likely to) contain encoding objects for both #My-Sequence-Class and for #SEQUENCE. Indeed, to make this easier, encoding object sets that are actually applied to perform encodings are usually constructed by the syntax:

```
ENCODED BY My-Encoding-Object-Set
    COMPLETED BY PER-BASIC-UNALIGNED
```

(Use of "PER-BASIC-UNALIGNED" is just an example. It could just as well have been "DER", or "PER-CANONICAL-ALIGNED".)

The "COMPLETED BY" forms what is called a **combined encoding object set**. This set contains all the encoding objects in "My-Encoding-Object-Set", plus any objects from the "PER-BASIC-UNALIGNED" set that are for encoding classes for which there is no encoding object in "My-Encoding-Object-Set".

A very important part of the above specification is that the encoding object for "#My-Sequence-Class" gets control before the determination of the encodings of the components. That encoding object is allowed to replace the entire structure, all components, all mandatory components, or all optional components, with a new

(parameterised) structure in which the existing structure/components are included as actual parameters. It is only after any such specified replacements that encoding proceeds to the components.

It is important to note here that if we have the situation that:

- Some of the repetitions in the structure are named as "#My-repetition-encoding" (a synonym for "#REPETITION");
- Others are named as "#SEQUENCE-OF" (also a synonym for "#REPETITION"); and
- The applied encoding object set includes an encoding object for "#My-repetition-encoding" but also the (basic unaligned, say) PER set of encoding objects;

then the encoding object for "#My-repetition-encoding" will be applied to all occurrences of that class, and basic unaligned PER encoding objects will be applied to the "#SEQUENCE-OF" classes.

It is possible to "color" (see below) an implicitly generated structure, leaving some of the implicitly generated "#SEQUENCE-OF" classes unchanged, but changing others to "#My-repetition-encoding". This enables some ASN.1 sequence-of constructions to be encoded using standard encoding rules, whilst others are encoded using specialized encodings.

This of course generalizes to the case where there are multiple different specialized encodings needed for the ASN.1 "SEQUENCE OF" construction.

3.9 Explicitly generated encoding structures

3.9.1 Introduction

Oh dear! It was all supposed to be done by implicitly generated structures. Now we have to learn how to explicitly generate structures.

An explicitly generated encoding structure is one which is created in (and exported from) an EDM module by importing and "coloring" an implicitly generated encoding structure from an ASN.1 module.

The explicitly generated encoding structure can be used within the EDM module that generates and exports it, but as a matter of style (and for simplicity and clarity) it is better if this is not done.

The ECN syntax is such that generation of an explicitly generated encoding structure causes its automatic export. This is important, because such a structure is only of use if an ELM can import it and apply encodings to it.

The explicitly generated encoding structure has the same simple class name as the implicitly generated encoding structure, but its fully-qualified name includes the module name of the EDM module that generated it. The **implicitly** generated encoding structure has a fully-qualified name that includes the name of the ASN.1 module in which the corresponding ASN.1 type was defined.

It is possible (but would be unusual) to define multiple explicitly generated encoding structures for an ASN.1 type (strawberry flavor, raspberry flavor, along-side the vanilla flavor of the implicitly defined encoding structure). Each of these different flavors could be imported into an ELM and encodings applied to it. These would produce different encodings for the original ASN.1 type. (There is, however, a rule that an ELM is not allowed to apply encodings to structures that have been generated from the same ASN.1 type.)

This simply reinforces the point that the encoding of an ASN.1 type is the encoding of a corresponding structure. The ELM determines (by importing it) the encoding structure to use, and the association between that structure and the ASN.1 type determines the type that the ELM is defining encodings for.

3.9.2 Defining an explicitly generated encoding structure

An explicitly generated encoding structure is defined by importing an implicitly generated encoding structure from an ASN.1 module, "coloring" it, and then exporting the result. The entire operation is performed in a "GENERATES-AND-EXPORTS" clause in an EDM, which appears between any "EXPORTS" clause and any "IMPORTS" clause.

As a matter of style, and for human readability, it is recommended that an EDM module performing "coloring" contains **only** a "GENERATES-AND-EXPORTS" clause.

What then is "coloring"? Coloring is the replacement of some class names in an implicitly generated structure with different class names **that are required to have been defined as synonyms for the names being replaced**.

The purpose is to indicate which parts of the structure are to be encoded using specialized encodings.

The classes being replaced can be any of the classes appearing in the structure, but will most commonly be a concatenation, optional, identifier, repetition, or alternative category of encoding class.

Real-world examples of coloring can be found in the ETSI ECN Case Studies. Here is a small example:

```
EDM-coloring-module
{joint-iso-itu-t(2) To be supplied}
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS My-encodings;
   -- An object set containing all specialized encodings
GENERATES-AND-EXPORTS
   REPLACING
      #OPTIONAL WITH #My-Optional
         IN ALL
   REPLACING
      #SEQUENCE-OF WITH #My-Sequence-of
         IN ALL EXCEPT #Group-Identity-Uplink, #Proprietary
   REPLACING
      #SEQUENCE WITH #My-Sequence
         IN #Group-Identity-Location-Demand
         EXCEPT #Group-Identity-Location-Demand.gia
   FROM My-ASN1-module {joint-iso-itu-t(2) ..... };

#My-Optional          ::= #OPTIONAL
#My-Sequence-of       ::= #SEQUENCE-OF
#My-Sequence          ::= #SEQUENCE

my-opt-encoding #My-Optional ::= ......
      -- An encoding object for the #My-Optional class

my-sequence-of-encoding #My-Sequence-of ::= ......
      -- An encoding object for the #My-Sequence-of class

my-sequence-encoding #My-Sequence ::= ......
      -- An encoding object for the #My-Sequence class

My-encodings #ENCODINGS ::= { my-opt-encoding |
                              my-sequence-of-encoding |
                              my-sequence-encoding }
   END
```

The module automatically exports the "colored" structure. It also defines encoding objects for the specialized encoding classes it has inserted into the structure (using syntax to be discussed below)), and forms these into an encoding object set (the last assignment statement) and exports this for use in an ELM. There it will (probably) be applied to the top-level structure in the set of explicitly generated structures that have been generated and exported using "GENERATES-AND-EXPORTS".

The "GENERATES-AND-EXPORTS" clause may have several comma-separated clauses terminating with "FROM", but this would be unusual.

The above definition imports **all** the implicitly generated structures from the ASN.1 module "My-ASN1-module" (there is one for each type assignment in the module), and generates and exports explicitly generated structures for **all** of these. The explicitly generated structures have the same simple names as the implicitly generated structures, but the fully-qualified names now include "EDM-coloring-module". An ELM wishing to import one of them (in order to define the encoding of top-level messages in the ASN.1 module) will import from the "EDM-coloring-module".

The above "GENERATES-AND-EXPORTS" specification identifies locations for changes, on what is effectively a textual basis. It references either the entire set of implicitly generated encoding structures from the

ASN.1 module (use of "ALL"), or names a single implicitly generated encoding structure, or names a component of that structure. The word "EXCEPT" is used in the obvious way, with either "ALL" or a list of one or more locations before the "EXCEPT", and a list of one or more locations after the "EXCEPT".

3.10 Defining encoding rules, parameterization, and replacement

Warning: This section gets a bit heavy! Wet-towels and lots of coffee are in order!

The intention here is to illustrate the dependence of ECN on parameterization and replacement mechanisms, when rules for encoding arbitrary ASN.1 are needed. Some of the examples given here use syntax that is discussed and introduced more fully later, and this makes the sub-clause a little heavy. However, the intent of the syntax is described, and the reader should concentrate on understanding the use of parameterization and of replacement, not on the syntax for individual definitions.

It is relatively easy to define encodings for a given structure, but in most real situations, a particular style of encoding (for example, use of an immediately preceding "presence-bit" mechanism for an optional element) is something that occurs in many places in a protocol, but with (in the above example) a different type as the optional component on each occurrence.

This proves to be slightly more complex.

What is needed is to "color" instances of such #OPTIONALs (as described above), so that they become #My-Optional, and then to define an encoding object for #My-Optional that will specify the required encoding for **any** optional component. This means parameterization of the specification, with a parameter that is the class of the optional component. This is exactly the same concept as in ASN.1, but it is heavily used in real ECN work.

We first define a parameterised structure that contains the presence bit with the component:

```
#New-component {< #Any-class >} ::=
    #SEQUENCE {
      presence-bit  #BOOL,
      component     #Any-class #My-Inner-Optional}
```

(In fact, if we miss out the presence-bit field, use a normal #OPTIONAL, and apply basic unaligned PER encodings to #New-component, we will get the desired effect, and everything is a **lot** simpler, but for the purposes of this tutorial, that would be cheating! So fill up your cup of strong coffee again!

Note that the syntax (see above) for parameter lists in ECN is "{<" and ">}", not the "{" and "}" used in ASN.1. This is to resolve ambiguities resulting from the ability to have synonyms for #SEQUENCE etc.

The #New-component structure can be used to replace any optional component that we wish to encode this way.

The second key feature is to get that replacement done automatically when we apply encodings.

We need eventually specify an encoding object for "#My-Optional", requiring it to:

- First replace the subject of the "#My-Optional" with the structure "#New-component", instantiating it with an actual parameter which is the subject of the "#My-Optional".
- Then to apply encodings to the instantiation.

This ability to require (if you wish to do so) encoding objects for constructors to perform replacements applies to all constructor classes, and is very powerful. In the case of the concatenation category of class, the replacement can be of all optional elements, or of all mandatory elements, or both (with different replacements for the two cases if required). In all cases it is also possible to specify replacement of the entire structure.

(There is also the ability to specify "head-end insertion" of additional fields in the case of a concatenation, but that goes beyond the scope of this tutorial.)

Whenever replacement takes place, the replacing structure is required to have a class parameter, and the instantiation uses the replaced structure as the actual parameter, thus preserving the abstract values that were present in the original structure.

The encodings to be applied to the instantiation are the ones being applied globally, plus an encoding object for the instantiation of "#New-component". We need to define the latter, and to supply that in the definition of the encoding object for "#My-Optional".

Our last actions in this clause will be to define that encoding object. As it is an encoding object for a parameterised class, it has, of course to be parameterised. (The encoding object for "#My-Optional" will instantiate it with the same parameter as it provides for the instantiation of the "#New-component" structure.)

But .... yes, a **second** parameter is required. In defining the encoding object for the "#New-component" structure, we need to encode all the components, but the encodings needed for the "#Any-class" component will be the combined encoding object set being globally applied in the ELM (which contains the encoding object for "#My-Optional"), so we need an "#ENCODINGS" parameter as well. The end-result is shown below.

For the encoding object for "#My-Inner-Optional" we define:

```
my-inner-opt-encoding {< REFERENCE: presence-bit-parm >} ::=
    { DETERMINATION MECHANISM aux-determinant
      DETERMINED BY presence-bit-parm }
```

This says that the optionality will be resolved using a determinant which is an auxiliary field that carries no application semantics. (The field here is assumed to be a boolean, with TRUE meaning present. We can add additional syntax to reverse that, or to use an integer field, but this part of the tutorial is getting over-weight anyway, so let's keep it simple!)

The actual determinant field is the dummy parameter "presence-bit-parm".

For the encoding object for "#New-component" we define:

```
new-component-encoding {<
          #Any-class,
          #ENCODINGS: Global-encodings >}
#New-component {< #Any-class >}
    ::=
    { ENCODE STRUCTURE
        { component
              OPTIONAL-ENCODING
              my-inner-opt-encoding
                {< presence-bit >} }
      COMPLETED BY Global-encodings }
```

This says that the encoding of #New-component is the encoding of the optionality of "component" by "my-inner-opt-encoding" with the actual reference parameter "presence-bit" (identifying a component of #New-component), and with all other encodings provided by the encoding object set parameter "Global-encodings".

We supply #New-component and "new-component-encoding" in the definition of an encoding object for #My-optional (including this in the encoding object set to be applied in the ELM), and we are done!

Let's go the last mile:

```
encoding-for-my-optional #My-optional ::=
    { REPLACE COMPONENT WITH #New-component
      ENCODED BY new-component-encoding }
```

A nice simple ending!

3.11 Differential encode/decode actions

You can't do it, but others can! And you can't even understand what they have written! Rotten!

Extensibility has been discussed earlier. There are many, many different approaches. But they **all** rest on the principle that when decoding you have to be prepared to accept (and to skip) things (usually random garbage!) that is different from what you are required to produce.

Here are some cases that arise:

- A reserved field, where version 1 is required to set it to zero, but to accept (and usually to ignore) any values in it.
- "Tail-end" optionals, where a system is required to ignore anything after the decoding of the known type is finished, and to accept absence of known tail-end optional elements. (In the simplest case a tail-end optional can only be included if earlier elements are included).

- Use of a TLV (Type, Length, Value) style of encoding, with instructions to skip and ignore elements with unknown types.
- Embedded extensions, with one or more length fields to enable versions ignorant of the extensions to skip the embedded material.

The bottom line, however, is that rules for encoding are different from those for decoding. Typically the encoder has a fixed specification of what to encode, but a decoder is told to assume that the sending encoder is operating to a different specification which allows random material (perhaps with a length wrapper mechanism known to all versions) to be included.

In ECN, this is called "differential encode/decode", and is supported at the highest level. For any encoding class, it is possible to define encoding objects (the reader must have got **that** message by now!). However, for any class it is also possible to define a differential encode/decode object that is simple a **pair** of encoding objects (defined in any of the many ways ECN provides). The first of the pair are the rules for encoding the class, and the second of the pair are the rules that a decoder should assume were used by a sender in encoding that class.

The classes to which differential encode/decode are applied invariably carry no application semantics - they are simply padding to an early implementation. The use of differential encode/decode is normally associated with the presence in a structure of #EXTENSIONS and #VERSION-BRACKETS, but it can be useful even when the ASN.1 specification does not use an extension marker, and relies entirely on differential encode/decode in ECN to specify its extensibility mechanisms.

This approach is, however, generally not recommended, because it is desirable for the application code to be aware of places where extensions are likely to occur in later versions. It is also frequently the case that extensions occurring in some places are "critical" and extensions occurring in other cases are "non-critical". This is best formally flagged by the ASN.1 exception marker on the extensions ellipsis, but of course human text can be used to describe these places.

3.12 Other fundamental concepts

This tutorial is too long already, but there remain some additional concepts. These are, however, very much for advanced use. Text may be added to this section in a later version of the tutorial, but for now there is just a list of further features:

- The #TRANSFORM class and its uses. (This allows, among other things, fields to be incremented or decremented or scaled-up before being used as determinants.)
- The #OUTER class. (This provides control over the padding of an outer-level encoding to – for example – an octet boundary, and the setting of reference points for alignment.)
- Using identification handles. (This generalises the BER “tag” concept to enable parts of an encoding to provide identifications that can act as choice or presence determinants.)
- Encoding objects that examine the bounds of integers or repetitions. (This enables a single encoding object to provide different encodings, depending on whether the actual instance of its use contains only bounded integers or contains unbounded integer values.)
- When fully-qualified names are needed. (This is detailed “scope rule” issues, and arises partly because there are some names allowed for ASN.1 types which, when a “#” is put in before them, become the same (reserved) names as are given to some built-in encoding classes.)

4 The process of defining a legacy protocol

Examples of this process can be found in the ETSI ECN Case Studies, and are not provided here. This clause simple describes the steps, and gives some good advice!

*I **hate** good advice, and **never** take it!*

4.1 Writing the ASN.1

Surprisingly, it is not easy to write good ASN.1 for a legacy protocol. (But I suppose some might say it is not easy to write good ASN.1 full-stop - there are some bad examples around!)

The choice of names for fields and types has to be a balance between reasonable brevity and maintaining a good link to the legacy protocol specification.

The most important decision, however, is what fields to make visible in the ASN.1. Writing the ECN can be easier if **all** fields in the encoding are reflected in ASN.1 components, but this is usually a **bad** design. The ASN.1 should normally only contain components for encoding fields that carry application semantics, not for auxiliary fields needed only for encoding purpose and providing presence and length determination, etc.

Additional fields can be added using ECN replacement mechanisms where necessary.

This point is quite important, because ASN.1+ECN tools will provide a, say, C or C++ or Java API that contains the components in the ASN.1 definition, delimiting them (or indicating choices/unions) in a language specific way. Having the same delimitation present in other parameters of the API just confuses the implementor, makes for larger implementation code, and increases testing complexity.

The increased testing complexity arises because there needs to be a check that the application code has correctly set the length component to the length of the component actually sent for encoding, rather than leaving the setting of the length field in the encoding entirely to the - hopefully tried and trusted! - encoding tool.

The ASN.1 should not be contorted to aid encoding, but suppose the legacy protocol specifies that:

- There are mandatory elements followed by optional elements.
- At the end of the mandatory elements there is an "options bit" that says whether at least one of the optional elements are present.
- If the "options bit" is set, each optional element has its own presence bit.

In this case, producing the ECN specification will be considerably eased if the optional elements are enclosed within an optional SEQUENCE, rather than simply being optional elements within the main PDU. The presence or absence of the optional SEQUENCE can then be used to generate the "options bit" that is required in the encoding.

Decisions of this sort are invariably somewhat subjective - there can never be one right piece of ASN.1 to describe a legacy protocol, but some can be said to be better than others!

4.2 "Coloring" the ASN.1

So we have written our ASN.1 specification.

Where we have some integers that are to be encoded one way (perhaps in binary) and some another (perhaps with BCD), we can reasonably identify these using different types both defined as "INTEGER":

```
Binary-integer      ::= INTEGER

BCD-integer         ::= INTEGER
```

and use those types in the body of the specification. (A purist would argue that this is bad ASN.1 - it contains encoding information, but let's leave that for the moment.)

We can then define encoding objects for "#Binary-integer" and for "#BCD-integer", and all is done.

However, in the case of different encodings for optionality, or for sequence-of, we have no mechanism in ASN.1 to define synonyms for these constructors. You can't write:

```
My-sequence-of      ::= SEQUENCE OF
```

(But of course you **can** do this in ECN, as we have seen earlier.)

The problem is over-come by "coloring" the implicitly generated encoding structure to produce an explicitly generated encoding structure in which the different "flavors" of sequence-of or optional encoding are clearly distinguished. Doing this "coloring" was described earlier.

It is normally good practice to reserve an EDM module to provide only the coloring, and to define and export the necessary class assignments such as that above.

For a small specification, however, it may be appropriate to include in that same EDM module the definition of the encoding objects for each of these specialized classes.

Notice that if you are a purist, and don't like defining "Binary-integer" and "BCD-integer" described above, you can leave all fields as "INTEGER", and use "coloring" to selectively change "#INTEGER" to "#Binary-integer" or to "#BCD-integer".

Finally, while it may be "cheating" (and we avoided doing it in an example above), you should use your knowledge of PER encodings and apply PER (probably the unaligned version) wherever possible. This will generally reduce the size of an ECN specification, but is also likely to speed up encode/decode processing with most ASN.1+ECN tools.

4.3 Defining the necessary encoding objects and object sets

The third step is to define the encoding objects for the specialized encodings needed.

There are many different ways of doing this, but most will probably involve some form of replacement of a class with another more complex class and then encoding that.

The later parts of this tutorial describes the different ways of defining encoding objects, and there are many examples in the Examples Annex of the ECN Standard.

Depending on the size of the task, the encoding objects can be grouped into a single EDM module, or can be defined in separate modules and exported.

You will generally want to produce an encoding object set containing all your specialized encodings, probably "COMPLETED WITH" the PER-basic-unaligned encoding object set, ready for application in the ELM. Alternatively, you can simply import each encoding object into the ELM and build the set in-line when you apply it. This would normally only be sensible if the ELM contains just a single encoding application.

4.4 Applying the encodings

The final step is to write your ELM module and to clearly identify it in the text describing the application. The ELM module has been designed to be simple and short, but provides (as its name implies) links - perhaps indirectly - to all other modules involved in the application specification. An example of an ELM module is given later in this tutorial, and in the Examples Annex of the ECN Standard.

5 How to define encoding objects

5.1 Model for defining simple bit-field encoding objects

Encoding-space and value-encoding are the key terms, but there are many, many other things to be concerned with!

Much of this tutorial so far has concentrated on defining special mechanisms for constructors and for #OPTIONAL. In this sub-clause we are concentrating primarily on defining (special) encodings for things like #BOOL, #INT, #BITS, and so on. Definition of encoding objects for constructors play a dual role. They are primarily concerned with the replacement of components, and with procedures for termination of repetitions, identification of alternatives, determination of optionality, and so. However, they also wrap-up the encoding of components to produce a bit-field in the encoding, and some of the functionality described below for simple bit-fields (for example, pre-alignment) is also present for constructors.

A picture is worth a thousand words. See the figure on the next page.

In defining simple encoding objects, values have to be given for a number of key encoding parameters that together totally define the encoding. It is the choice of these key encoding parameters that determines the "quality" of ECN in this area, and whether it is capable of specifying the encodings that a user wants. (The use of the expression "encoding parameter" here should not be confused with the earlier use of "parameterization". The use here is normal English.)

To understand this part of ECN, the available encoding parameters have to be understand, and many of them relate to the above figure. In broad terms they break down into two sets of encoding parameters:

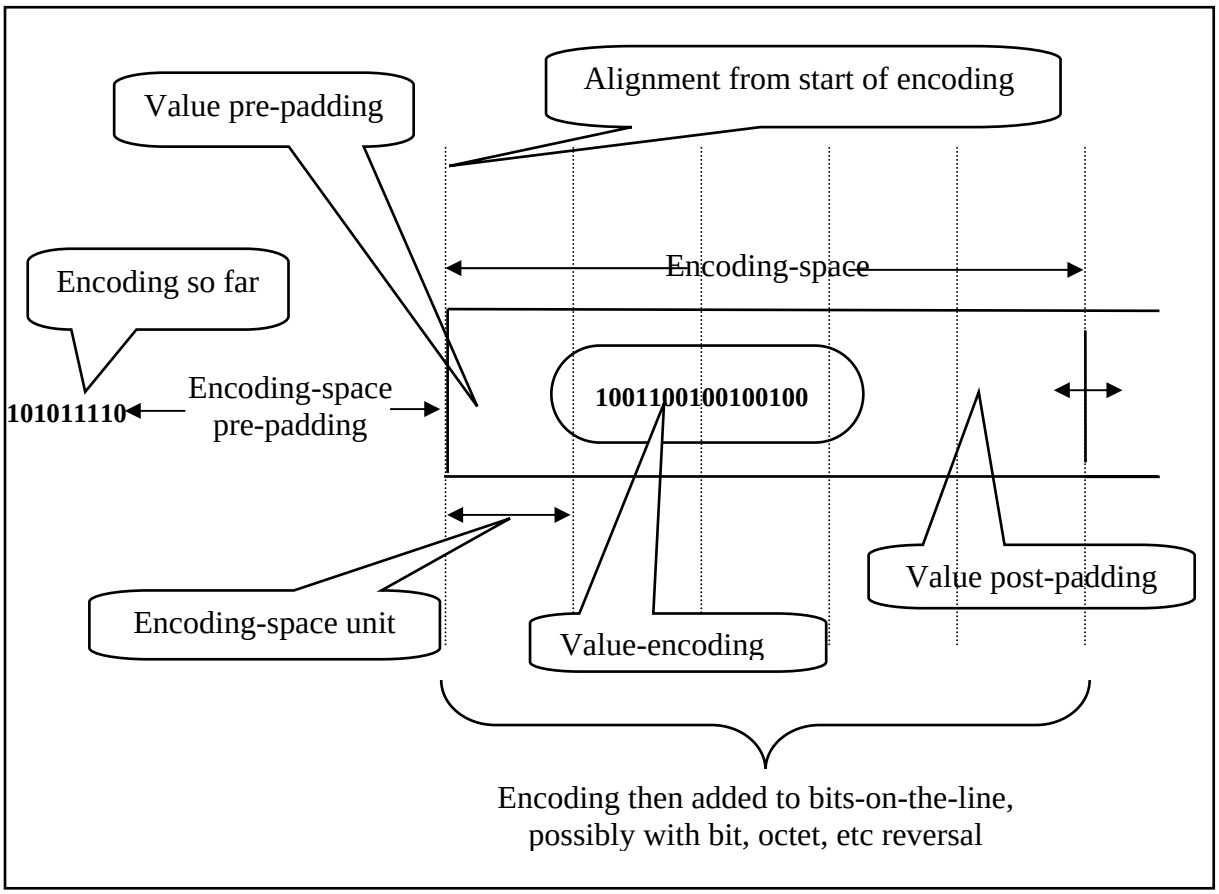
- Encoding parameters concerned with the amount of space the value will encode into (including the units of space allocation and how the end of a variable length space is to be determined).
- Encoding parameters concerned with the actual bit-patterns used to represent values associated with the class.

These are called, respectively, definition of the **encoding-space** and definition of the **value-encoding**.

The set of encoding parameters for defining the encoding-space is in most cases both the larger set of encoding parameters and the most complex! However, these encoding parameters are generally common to the definition of encoding objects of most classes, whereas the set concerned with value-encoding tend to be different for each class.

The distinction between these two sets of encoding parameters is much less clear today than it was in early work, and there are also encoding parameters associated with the replacement of components by other structures that fall into neither category.

In all cases where bits are being generated, there are the following activities that can be specified:



- Specification of so-called **pre-padding**. This provides for alignment of the start of the encoding space to (for example) a byte or word boundary, and the pattern of any padding that is inserted.
- Specification of the encoding-space unit (the encoding space is always a multiple of this unit).
- Specification of the value-encoding.
- Specification of the positioning of the value-encoding (typically right or left justified) in the encoding space, with the pattern of any padding that has to be inserted.
- Finally, there are encoding parameters to determine whether reversal of octets or of bits within an octet is to occur before the total encoding is passed for transmission.

The model focuses primarily on rules for encoding. Decoders have to work out for themselves how to decode such material, but in most cases it is quite obvious from the encoding specification.

Alignment to word boundaries (for example) counts bits "from the start of the encoding". There is one exception to this: if the ASN.1 contents constraint (introduced by “CONTAINING” and “ENCODED BY”) has been applied to a bit string or octet string type, then alignment in the contents encoding can either be relative to the start of the bit or octet string, or relative to the start of the encoding. This is controlled by the encoding object applied to the bit or octet string.

A question to the reader - you have heard of defining Information Object Classes, and the "WITH SYNTAX" clause? This is nothing to do with ECN, it is a feature added to ASN.1 in the early 1990s, but not all users of ASN.1 are familiar with it. The definition of encoding parameters for defining encoding objects for simple

classes, and the definition of the syntax to be used in such definition uses a very similar notation. But do not despair - if you have never met these concepts, the notation is fairly obvious and user-friendly!

The set of encoding parameters for defining encoding objects of simple bit-field classes (and also for constructors) are specified for each class in Annex A of the ECN Standard. Each class has roughly a page of definitions of the encoding parameters available to define it, using a notation for defining these encoding parameters that is a minor variation of the syntax used to define Information Object Classes. A reader wishing to understand the Annex A specification needs to have a reasonable understanding of Information Object Classes. (A description of each parameter is given in clause 19 of the ECN Standard, and readers wanting more information should consult that clause.)

Each Annex A specification ends with a "WITH SYNTAX" clause which defines the syntax to be used for specifying values of the encoding parameters. Again, the notation used in defining this syntax is that which ASN.1 users use in specifying a "WITH SYNTAX" clause in an Information Object Class definition. Formally, defining encoding objects using this syntax is called using **defined syntax** for the definition of an object.

An important feature of the Annex A specification is the extensive use of default values, which generally means that most of the time the majority of encoding parameters can be omitted.

Here is the definition of encoding parameters for defining an encoding object of class #BOOL, copied from the ECN Standard. Don't worry if you do not understand it! Just get the flavor! Clause 8 of this Tutorial states the intention to provide tutorial material (in a subsequent version) for each of the specifications in Annex A. That text will describe the syntax in an informal manner, and knowledge of the formal notation shown below will not be assumed. The definition for #BOOL is:

```
#BOOL ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding             Padding DEFAULT zero,
  &encoding-space-pre-pattern             Pattern (ALL EXCEPT other)
                                          DEFAULT bits:'0'B,

-- <Bool value-encoding>
  &value-true-pattern                     Pattern DEFAULT bits:'1'B,
  &value-false-pattern                    Pattern DEFAULT bits:'0'B,

-- <Encoding space>
  &encoding-space-unit                    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                    Size (uses-determination-mechanism | fixed-to-max |
                                                  1..MAX) OPTIONAL,

-- <Padding>
  &value-justification                    Justification DEFAULT right:0,
  &value-pre-padding                      Padding DEFAULT zero,
  &value-pre-pattern                      Pattern (ALL EXCEPT other)
                                          DEFAULT bits:'0'B,
  &value-post-padding                     Padding DEFAULT zero,
  &value-post-pattern                     Pattern (ALL EXCEPT other)
                                          DEFAULT bits:'0'B,

-- <Determinant>
  &encoding-space-length-determinant      REFERENCE OPTIONAL,
  &Encoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".

-- <Primitive handle information>
  &handle-id                              PrintableString OPTIONAL,
  &Handle-positions                       INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                           ReversalSpecification
                                          DEFAULT no-reversal  }

WITH SYNTAX {
      [ENCODING-SPACE
```

```
        [ALIGNED TO &encoding-space-pre-alignment-unit
                [PADDING &encoding-space-pre-padding
                        [PATTERN &encoding-space-pre-pattern]]]
        [SIZE &encoding-space-size]
        [MULTIPLE OF &encoding-space-unit]
        [DETERMINED BY &encoding-space-length-determinant
                [ENCODER-TRANSFORMS &Encoder-transforms]]]
    [VALUE      [TRUE-PATTERN &value-true-pattern]
                        [FALSE-PATTERN &value-false-pattern]
            [JUSTIFIED &value-justification]
            [PRE-PADDING &value-pre-padding
                [PATTERN &value-pre-pattern]]
            [POST-PADDING &value-post-padding
                [PATTERN &value-post-pattern]]]
        [HANDLE &handle-id AT &Handle-positions]
        [BIT-REVERSAL &bit-reversal]
```

}

Now let us use that notation to completely define an encoding object for #BOOL. (We are on page 20+ of the tutorial and this is your first example of an actual encoding object definition - oh dear!)

Our first example will encode into a single bit (with no alignment), but just to be awkward, we will use zero for true and one for false! We get:

```
my-bool-encoding #BOOL ::=
    {ENCODING-SPACE
        SIZE 1
        MULTIPLE OF bit
    VALUE
        TRUE-PATTERN  bits:'0'B
        FALSE-PATTERN bits:'1'B }
```

Now suppose we want some or our booleans encoding into a single octet, starting on an octet boundary (padding with zeros if necessary - the default padding, so we don't specify it). This time we want an encoding of zero for false and any-non-zero octet for true (similar to BER). We "color" our structure with "#BOOL2" (defined below as a synonym for #BOOL), and write:

```
#BOOL2 ::= #BOOL

my-bool2-encoding #BOOL2 ::=
    {ENCODING-SPACE
        ALIGNED TO octet
        SIZE 1
        MULTIPLE OF octet
     VALUE
        TRUE-PATTERN octets:'00'H
        FALSE-PATTERN other }
```

Our encoding object set can contain both "my-bool-encoding" and "my-bool2-encoding", and dependent on the coloring, one or other will encode boolean fields.

5.2 Mechanisms for defining encoding objects

Seven mechanisms - what a surprise! The magic seven comes up yet again!

There are seven mechanisms available for defining encoding objects of some class. Use of "defined syntax" described above is just one of these seven mechanisms. We briefly discuss and illustrate each mechanism below.

5.2.1 Use of an already-defined encoding object name

Very simple things are sometimes very powerful, but this one isn't!

Suppose we have defined "my-bool2-encoding" as above. We can define a new encoding object with a different name by writing:

```
second-bool2-object #BOOL2 ::= my-bool2-encoding
```

The class of the object to the right of the "::=" has to be the same as the governor, so this rather simple mechanism is no big deal!

5.2.2 Use of "defined syntax"

An ECN sub-language: Designed to be easy to read! Is it?

"Defined syntax" has been extensively discussed and illustrated above. A later version of this tutorial will provide a detailed description of each of the "Annex A" defined syntaxes, their keywords and their functionality.

The Annex A syntax is, however, still a little unstable, with some of the keywords and keyword combinations not reading too well, so effort has not been put into writing a tutorial for this material at this time.

5.2.3 Use of an existing encoding object set

Sub-titled: How to get your hands on a PER encoding object!

We can write:

```
per-int-encoding #INTEGER ::=
    { ENCODE WITH PER-BASIC-UNALIGNED }
```

or

```
per-structure1-encoding #Structure1 ::=
    { ENCODE WITH DER }
```

This syntax is illustrated using standardized encoding object sets, but any object sets can be used, including if necessary a "COMPLETED BY" clause (illustrated earlier) to form a combined encoding object set to be applied to the class for which we need an encoding.

5.2.4 Use of value mappings

*Now here *is* power! But there are several variants, and we need another clause to cover them all!*

This is a powerful mechanism with many applications. We want an encoding object for an existing class (perhaps a simple "#INTEGER" class, or perhaps a more complex class - call it "#Old-class"). This class does not contain all the fields that we need in our encoding (perhaps it was an implicitly generated class for some part of our ASN.1 definition).

We can proceed as follows:

- Define a new structure that contains all the fields in our required encoding (call this "#Replacement-structure").
- Specify the way values of "#Old-class" map into values of "#Replacement-structure", using value mapping syntax described below.
- Specify that encoding of values of "#Old-class" is the encoding of the corresponding value of "#Replacement-structure", encoded by some encoding object for the class "#Replacement-structure".

In this case, we will be a bit clever, and instead of defining and naming an encoding object of class #Replacement-structure for the last bullet, we will define it in-line using the "ENCODE WITH" syntax described above.

We write:

```
encoding-for-old-class #Old-class ::=
    { USE #Replacement-structure
      MAPPING
      ......
      WITH
       { ENCODED WITH PER-BASIC-UNALIGNED }
       -- The above line is defining an encoding object
       -- for the class #Replacement-structure
```

There are a number of pieces of syntax that can be used to perform value mapping between two structures. These are described in Section 5.3.

None of these pieces of syntax work for arbitrary and totally dissimilar structures of "#Old-class" and "#Replacement-class", but there is syntax to support most value mappings between two structures that can be considered "obvious".

5.2.5 Encoding a structure component-by-component

Divide and conquer!

This mechanism enables an encoding object to be defined for a class that is an encoding structure. The reference name for the structure is de-referenced to give the definition of that structure, but no further de-referencing takes place. The encoding is specified by listing any (or all) of the field-names which are now visible in the structure and specifying encodings for those fields, and for the constructor at the head of the structure. (The mechanisms also apply to a structure that is headed by a repetition category of class).

Suppose we have a structure:

```
#Sequence1 ::= #SEQUENCE {
    a           #Structure1,
    inserted      #BOOLEAN,
    b           #INTEGER #OPTIONAL,
    c           #OCTET-STRING }
```

Then we can define an encoding object for it with:

```
sequence1-encoding #Sequence1 ::= {
    ENCODE STRUCTURE {
     b OPTIONAL-ENCODING
         {ENCODING-SPACE AS aux-determinant
          DETERMINED BY inserted } }
    WITH { ENCODE WITH PER-BASIC-UNALIGNED } }
```

Here we are applying a special encoding for the optionality of "b", referencing field "inserted" as the determinant. Everything else is encoded with "PER-BASIC-UNALIGNED". We could also have specified a specialized encoding object for the #INTEGER in "b", and/or for fields "a", "inserted", and "c", and/or for the #SEQUENCE constructor itself (the latter specification would include the keywords "STRUCTURED WITH").

This mechanism provides a flexible way of defining encoding objects for a general structure, and would usually be combined with the definition of value mappings from an old structure to a new structure that had additional auxiliary fields in it (for example the "inserted" field in the above example).

5.2.6 Differential encode/decode object definition

Now you have two lots of encoding rules to worry about!

We have already discussed the concept of differential encode/decode objects. These are simply the combination of two encoding objects, one specifying the encoding to be used, the other specifying the encoding to be assumed when decoding. The syntax is fairly straight-forward:

```
object-for-tail-end-additions  #PAD ::=
    { ENCODE-DECODE
         {ENCODING-SPACE SIZE empty}
         -- Specifies an empty encoding for ENCODE
      DECODE AS IF
         {SIZE uses-determination-mechanism
          DETERMINED BY container
          CONTAINED IN end-of-encoding:NULL
          PADDING encoder-option } }
```

Here we specify both the encoding object and the assumed decoding object in-line. For encoding, the #PAD field (assumed to be placed at the end of the PDU) is encoded as an empty string (no bits). For decoding, the decoder assumes an encoder has put arbitrary padding in the #PAD field, up to the end of the PDU. This is a common approach to provisions for extensibility, and formally specifies that implementations of this version of the protocol should ignore additional material at the end of the encoding up to the end of the containing PDU.

5.2.7 User-defined encoding-functions

User-defined encoding-functions allow an encoding object to be defined using any notation you wish! Very flexible, but probably not much use for standardized encodings. The chosen notation can even be comment.

Try:

```
encoding-of-funny-type #Funny-type ::=
    USER-FUNCTION-BEGIN
    {joint-iso-itu-t(2) etc } -- specifies the notation used
    /* Pseudo-code for encoding values of #Funny-type */
      ......
    /* Pseudo-code for decoding values of #Funny-type */
      ......
    USER-FUNCTION-END
```

A cop-out that let's ECN assert it can do anything!

Tool support for user-defined functions is likely to take the form of an escape from the standard encode/decode procedures to an application function that will provide code for the specified encoding.

There may be some notations that could be used in the user-defined function that have tool support (for example, CSN.1), but it should normally be possible to define any encodings that are really needed using the only formally-defined ECN syntax (which will have full integrated tool-support). This mechanism is an escape if ECN fails you in some detailed part of your specification.

5.3 Value mapping mechanisms

Here we look briefly at the syntax which is used for the value mappings discussed above.

Only six mechanisms - something has gone wrong! Bet there will be a seventh soon!

Recap: This notation appears in place of the "....." below:

```
encoding-for-old-class #Old-class ::=
    { USE #Replacement-class
      MAPPING
      ......
      WITH
       encoding-of-replacement }
```

It provides mappings of values from "#Old-class" to "#Replacement-class".

5.3.1 Mapping single values

This notation is available only if both "#Old-class" and "#Replacement-class" are classes implicitly generated from primitive ASN.1 types, and the mapping uses the ASN.1 value notation for those types. Quite simply, the notation is a list of pairs of values, one from "#Old-class" and one from "#Replacement-class". It can therefore map only a finite set of values. An example might be:

Detailed and tedious and quite restricted. But it's there if you find it useful.

```
encoding-for-old-class #UTF8String ::=
    { USE #INTEGER
      MAPPING VALUES {
        "0" TO 0, "1" TO 1, "2" TO 2, "3" TO 3,
        "4" TO 4, "5" TO 5, "6" TO 6, "7" TO 7,
        "8" TO 8, "9" TO 9, "*" TO 10, "#" TO 11 }
      WITH
       encoding-of-integer }
```

Where the mapping is from "#INTEGER" to "#BITS", syntax with slightly more functionality (ranges can be specified) is available - see below.

5.3.2 Mapping by matching fields

Here "#Old-class" and "#Replacement-class" have to be "similar". The syntax is intended for the case where "#Replacement-class" has all the same top-level field-names

You want to add a new auxiliary field? This is for you!

(with exactly the same class for the field) as "#Old-class", but they may be in a different order, and there may be additional fields which will eventually be used as determinants. The semantics are simple - values are mapped between fields with the same name, and the syntax is even simpler - just the keyword "FIELDS":

```
encoding-for-old-class #Old-class ::=
    { USE #Replacement-class
      MAPPING FIELDS
      WITH
       encoding-of-replacement }
```

5.3.3 Mapping using #TRANSFORM encoding objects

Advanced stuff! This is beyond your tutor!

This syntax provides very flexible mappings from integers to other integers by adding 1, dividing by 2, and so on. (Useful if determinants count in funny units, or have offsets). It also provides "transformations" (that are really encodings) of integers into bits or into characters, letting you later encode those bits or characters.

The #TRANSFORM class has not been covered in this tutorial, and further discussion of this mapping mechanism is abandoned. Version 2 of the tutorial may do better! Otherwise, go on an ECN course and see if your trainer can tell you all about mappings using #TRANSFORM objects!

For completeness, however, here is an example from the ECN Standard. We have an #Old-class which is #INTEGER, containing only even number values 0, 2, 4, etc, and we want to encode these as 0, 1, 2, etc

```
encoding-for-old-class #INTEGER1 ::=
    { USE #INTEGER2
      MAPPING TRANSFORMS
      { {INT-TO-INT divide:2} }
      WITH
       encoding-of-integer2 }
```

It would also be possible to achieve this by MAPPING ORDERED VALUES, described in the next sub-clause.

5.3.4 Mapping by abstract value ordering

***Very** flexible, can map an infinity of values between many different structures. Good for playing clever tricks with ECN!*

If you can order the abstract values of "#Old-class" and those of "#Replacement-class", even if those sets of values are infinite, then you can map the values between them.

Classes with a defined ordering of values are those derived from #NUL, #BOOL, #INT (provided there is a lower bound), #REAL (provided it is constrained to a finite set of values), #ENUMERATED **and any class that is constructed as an #ALTERNATIVE construction using classes with a defined ordering**.

The syntax is again very simple, just the words "ORDERED VALUES":

```
encoding-for-old-class #Old-class ::=
    { USE #Replacement-class
      MAPPING ORDERED VALUES
      WITH
       encoding-of-replacement }
```

It has been used to map a positive integer into a choice of bounded integers (which is then encoded using PER) to provide a Huffman-style encoding of those values. Huffman encodings are, however, better done using the last mapping in this section.

Whilst very powerful, this mechanism is again mainly for advanced users who want to do somewhat unusual things.

5.3.5 Mapping by distributing values

Similar to mapping ordered values, but with a little more control.

In this case #Old-class has to be an integer category of class, and the #Replacement-class has to contain one or more integer category of

fields. The user specifies which ranges of the original integer map into which of the integer fields in the #Replacement-class, which will typically be a structure using #ALTERNATIVES. Suppose #Replacement-class has integer fields "field1", "field2, and "field3" (it may also have other fields that will be used as determinants). We can write:

```
encoding-for-old-class #Old-class ::=
    { USE #Replacement-class
      MAPPING DISTRIBUTION
      0 .. 9 TO field1,
      10 .. 99 TO field2,
      REMAINDER TO field3
      WITH
       encoding-of-replacement }
```

5.3.6 Mapping integer values into bits

You control the encoding completely

This is intended for the definition of Huffman encodings of integer values. The #Old-class has to be an integer category of class and the #Replacement-class has to be a bits category of class. A true Huffman encoding of an integer needs as input the relative frequency with which each integer value will occur in a series of communications. There are algorithms that can take this information and produce self-delimiting encodings (no length determinant needed) of each integer value which produce the theoretical minimum of bits-on-the-line, averaged over a sufficient number of communications. Frequently occurring values use a small number of bits, infrequently occurring values a larger number. A Microsoft Word macro that will generate Huffman encodings from percentage frequencies is available on the ECN web-site at http://asn1.elibel.tm.fr/ecn. Here is an example of the use of the macro. The lines between "-- End Definition" and "WITH ...." were generated by the macro from the frequency information. An example of the final output, produced in ECN syntax, is:

```
encoding-for-integer #INTEGER ::=
    { USE #BITS
      -- ECN Huffman
      -- Range (-1 ..10)
      -- -1 is 20%
      -- 1 is 25%
      -- 0 is 15%
      -- (3..6) is 10%
      -- REST IS 2%
      -- End Definition
      -- Mappings produced by "ECN Public Domain Software for
      -- Huffman encodings, version 1"
      MAPPING TO BITS {
      -1 TO '11'B,
      ( 0 .. 1 ) TO ( '01'B..'10'B),
      2 TO '0000001'B,
      ( 3 .. 5 ) TO ( '0001'B..'0011'B),
      6 TO '00001'B,
      ( 7 .. 8 ) TO ( '0000010'B..'0000011'B),
      ( 9 .. 10 ) TO ( '0000000'B..'00000001'B)
      }
      WITH
       encoding-of-bits }
```

6 How to define encoding object sets

You know this already. Short and sweet.

Encoding object sets have names beginning with a capital letter, and the governor for set notation is the reserved name #ENCODINGS. Given encoding objects "obj1", "obj2", etc we can write:

```
My-encodings #ENCODINGS ::=
    { obj1 | obj2 | obj3 | obj4 }
```

Encoding object sets can be defined and named in an EDM, but in an ELM they can only be formed in-line (from imported encoding objects) when applying encodings.

As with similar ASN.1 notation, each of the constituents used in building an encoding object set can be either an encoding object or an already-defined encoding object set.

And there is little more to say here either - the end is nigh!

7 How to apply encodings in the ELM

The final step - we apply an encoding object set to a generated structure corresponding to an ASN.1 type definition. This may be the implicitly generated structure, imported from the ASN.1 module, but is more likely to be an explicitly generated structure imported from an EDM module.

The corresponding ASN.1 type will typically be the (usually single) top-level type that defines all the messages of the application. Typically that type contains many type references (and the corresponding encoding structure many encoding class references), but it is **not** necessary (nor meaningful) to apply encodings to all these constituent parts. When an encoding object set is applied it ripples down to all the remotest parts of the type tree, with all type references being resolved as it proceeds through the structure. Thus the one application fully defines the encoding of all messages. Here is a typical ELM module:

```
ELM-module-my-name
{joint-iso-itu-t(2) To be supplied}
LINK-DEFINITIONS ::=
BEGIN

IMPORTS My-encodings FROM EDM-Encodings-module { ..... }
     #My-Messages FROM EDM-Coloring-module { ..... };

ENCODE #My-Messages WITH My-encodings
    COMPLETED BY PER-BASIC-UNALIGNED

END
```

The ELM is a very lazy module - all the real work is done in the EDM's, but the ELM takes all the credit! It is only when encodings are applied in the ELM that actual definition of the encoding for values of an ASN.1 type occurs. All activity in the EDM is simple defining encoding objects for particular classes in order to produce the encodings of the ASN.1 type that are needed.

The ELM will typically contain precisely one "ENCODE" statement, but can contain more (and can even refer to all the types in a module if they represent separate messages), but there are rules that prevent the ELM from defining encodings twice for the same ASN.1 type.

8 Examples of simple encoding object definition

Oh dear, more and more still to read!

This part of the tutorial is incomplete in version 1. It will eventually copy material from the ECN Standard Annex E to illustrate and describe the definition of:

- Encoding objects for classes derived from #ALTERNATIVES
- Encoding objects for classes derived from #BITS
- Encoding objects for classes derived from #BOOL
- Encoding objects for classes derived from #CHARS
- Encoding objects for classes derived from #CONCATENATION
- Encoding objects for classes derived from #INT
- Encoding objects for classes derived from #NUL
- Encoding objects for classes derived from #OCTETS
- Encoding objects for classes derived from #OPTIONAL
- Encoding objects for classes derived from #PAD
- Encoding objects for classes derived from #REPETITION
- Encoding objects for classes derived from #TAG

For the present, the reader is referred to that Annex.

9 Encoding objects for commonly used encoding techniques

This is the "how-to" section - pity it hasn't been written yet!

It is intended that Version 2 of this Tutorial will describe in detail the way in which ECN can be used to support each of the following techniques.

This is intended to provide a library of useful definitions, and may be extracted into a separate document once produced.

The encoding techniques currently planned for inclusion are:

- An immediately prefixed presence bit for #OPTIONAL
- An M-bit (a “more” bit) for zero or more elements (1 before each element, zero after the last) in #SEQUENCE-OF.
- An M-bit for one or more elements (each element has an M-bit before it which says whether another element follows) in #SEQUENCE-OF.
- Bit length wrappers of W-bits for #SEQUENCE.
- Bit length wrappers of W-bits for #SEQUENCE-OF.
- Byte length count wrappers of W-bits for an aligned #SEQUENCE-OF with an unused bits determinant.
- Element count wrappers of W-bits for #SEQUENCE-OF.
- Huffman encoding of #INTEGER.
- BCD encoding for an #INTEGER using a PER #OCTET-STRING encoding and an even/odd determinant.
- Tail-end optionals terminated by the end of a #SEQUENCE with a length wrapper of W-bits.
- Tail-end optionals terminated by end of PDU.
- An encoding object for #TAG which produces a W-bit wide encoding without alignment.
- An encoding object for #TAG which produces a W-byte wide encoding with alignment.
- Differential encode/decode for #PAD parameterised by zero or one padding, and with the termination conditions selected from any combination of fixed, next n-bit alignment point, or end of PDU.
- Two sequence-ofs with the same length
- Encodings which use a list of pointers near the head of a sequence of elements, each pointer identifying the position of the start of the encoding of an element, with arbitrary padding between the end of one element and the start of the next.

Examples of some of these techniques can already be found in the Examples Annex of the ECN Standard, or in the ETSI Case Studies.

Suggestions for additional "useful techniques" that might be included are welcomed, and should be sent to the author at j.larmouth@salford.ac.uk.

10 Further reading and guide to the ECN Standard

The ETSI Case Studies *References to be provided*

The ECN Standard - *X.692 | ISO/IEC 8825-3 and ETSI .....*

If you have got this far, you will probably want a rest! But if you still have the energy, try these as follow-up!

There are several ETSI Case Studies in preparation, each one applying ECN to a legacy protocol. The application to TETRA is the most advanced, and it is expected that these Case Studies will become available progressively from December 2000. Readers

wishing to know about availability of these documents should subscribe to the ECN mailing list ecn@oss.com. (to subscribe, send an e-mail to majordomo@oss.com with "subscribe ecn your-name" in the body).

The final authority for those wanting to make serious use of ECN or to begin implementations is the actual ECN Standard ITU-T X.692 and ISO/IEC 8825-3. These are still in the form of (fairly complete) drafts, and have been fairly widely circulated. They will in due course become ETSI documents that are freely available.

As stated earlier, this tutorial is (as far as is possible) aligned with the functionality expected in the **next** version of the ECN Standard, due to be available mid-February 2001, and it is recommended that most readers wait for the availability of that document.

The ECN Standard itself contains some fairly brief tutorial material in its clause 9, a full description of the Annex A parameters in clause 19, and a very extensive Examples Annex. These are all useful follow-up reading. And, of course, it contains the authoritative and complete definition of the ECN syntax.