Semantix
INFORMATION TECHNOLOGIES

ESA/ESTEC Contract: 22259/09/NL/CBI

« ASN/1 Space Certifiable Compiler»

Deliverable D2

«ACN User Manual»

issued

2010-07-29

Document Ref:	ACN-UM-D2
Global Id:	SMX-SPACE-ESA-C22259-ACN-UM-D2
Version:	1.0
Editing tool:	Microsoft Word 2007
Authors:	George Mamais (gmamais@semantix.gr)
	Menelaos Perdikeas (mperdikeas@semantix.gr)
Reviewers:	Thanassis Tsiodras (ttsiodras@semantix.gr)

SEMANTIX Information Technologies S.A.
Konstantinou Tsaldari 62, 11476, Athens, GREECE
T: (+30) 210-6412065
F: (+30) 210-6412068
E-mail: info@semantix.gr
Web: www.semantix.gr

TABLE OF CONTENTS

TABLE OF CONTENTS 2

ACRONYMS AND ABBREVIATIONS 3

1. WHAT IS ACN? 4

2. SHORT INTRODUCTION TO ACN 5

2.1. Auto generation of default ACN grammar 6

3. ACN ENCODING PROPERTIES 7

3.1. size property 7
- 3.1.1. Fixed form 7
- 3.1.2. Variable size with embedded length 7
- 3.1.3. Variable size with length specified in external field 8
- 3.1.4. Null terminated 8

3.2. encoding property 8

3.3. adjust property 9

3.4. align-to-next property 10

3.5. encode-what property 11

3.6. true-value and false-value properties 11

3.7. optionality property 11

3.8. present-when property 11

3.9. determinant-uper and determinant-tag properties 12

4. ENHANCED OPTIONS 14

4.1. Fields introduced in the ACN grammar 14

4.2. Parameterized encodings and deep field access 14
- 4.2.1. Length determinant is below current node 14
- 4.2.2. Length determinant is above current node 15
- 4.2.3. Length determinant is in completely different subtree 16

ACRONYMS AND ABBREVIATIONS

ANSI	American National Standards Institute
API	Application Programming Interface
ASN.1	Abstract Syntax Notation 1
ASN1scc	ASN.1 space certifiable compiler
AST	Abstract Syntax Tree
BER	Basic Encoding Rules
EAST	Enhanced Ada SubseT
ECN	Encoding Control Notation
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
PER	Packed Encoding Rules
XML	eXtensible Markup Language
XSD	XML Schema Documentation

1. WHAT IS ACN?

ASN.1 is a language for defining data structures (i.e. messages) in an abstract manner. An ASN.1 specification is independent of the programming language (C/C++, Ada etc), the hardware platform or even the encoding method used to serialize the defined messages. The encoding mechanism, i.e. how bits and bytes are written over the wire, is determined by the ASN.1 encoding. Although the standardized ASN.1 encodings may offer some important benefits such as speed and compactness for PER or decoding robustness for BER, there is no way for the designer to control the final encoding (i.e the format at the bit level). This is a problem for situations where there is legacy binary protocol and we must replace one of the communicating parties using ASN.1 encoders/decoders (i.e. when the other, legacy system, must remain unchanged).

<u>ACN is a proprietary ASN.1 encoding which addresses the above need: it allows protocol designers to control the format of the encoded messages at the bit level.</u>

The main features of ACN are:

- Easy to learn, with simple and clear syntax but also with enough power to cover complex cases
- Encoding instructions are written in a separate file, so that the original ASN.1 grammar remains unpolluted
- Fields which do not carry semantic information but are used only during the decoding process (e.g. length fields, choice determinants etc) may either appear in the ASN.1 grammar or introduced only in the ACN specification.

The sections that follow showcase ACN through easy to follow code examples.

2. SHORT INTRODUCTION TO ACN

Every ASN.1 type has a set of encoding properties that can be set in order to achieve the desired binary encoding. These properties control certain aspects of the encoding process such as: the size of type being encoded, how values are encoded (twos-complement vs positive integer encoding, etc), the presence/absence of a certain field etc.

These properties are assigned to ASN.1 types using a pair of square brackets ("[" and "]") as seen in Listing 2. The encoding properties assignment is carried out in a separate file – the ACN file, so that the original ASN.1 grammar remains "clean" from encoding specifications.

Here is a simple ASN.1 grammar:

```
MYMOD DEFINITIONS AUTOMATIC TAGS::= BEGIN
  MyInt ::= INTEGER (-100 .. 100)
  MyInt2 ::= INTEGER (0 .. 1000)
  MySeq ::= SEQUENCE {
          a1 INTEGER (1..20),
          a2 INTEGER (-10 .. 20),
          a3 MyInt,
          a4 MyInt2
  }
END
```

Listing 1: Sample ASN.1 grammar

and here is an example ACN encoding for this grammar:

```
MYMOD DEFINITIONS ::= BEGIN

    --ACN allows constant definitions
  CONSTANT WORDSIZE ::= 32
    --We can make basic math with ACN constants
  CONSTANT LARGEST-INT ::= 2^^(WORDSIZE - 1)-1

  --MyInt will be encoded as twos complement integer.
  --Size will be 1 byte
  MyInt[size 8, encoding twos-complement]

  -- If no encoding properties are present, then
  -- encoding properties will be automatically populated
  -- so that the behavior matches the one of uPER i.e.
  -- size 10, encoding pos-int
  MyInt2 []

  -- encoding properties for types defined
  -- within constructed types (i.e. fields)
  MySeq [] {
          a1 [size auto, encoding pos-int],
          a2 [size 32, encoding twos-complement, endianness little],
          a3 [],
          a4 [size 11, encoding twos-complement] – allowed encoding range is -1024..1023
  }
END
```

Listing 2: Sample ACN grammar for the ACN grammar of Listing 1

By looking at the above code example, we see the following:

- For each ASN.1 module there is one ACN module with the same name.
- We can optionally define some integer constant values (WORDSIZE, LARGEST-INT etc) which can be referenced by the rest of the ACN specification.

- The ACN module contains the types (i.e. the type references) declared in the ASN.1 module followed by the encoding properties.
- The encoding properties may be absent. (The pair of open close brackets [ ] must be present though). In this case, the encoding properties have values which are calculated as follows:
  - Referenced types inherit the properties of their base types
  - For non referenced types (or referenced types whose base types have no encoding properties), the encoding properties are automatically populated with such values as to mimic the behavior of uPER.
- For types declared within constructed types such as `SEQUENCE` / `CHOICE` / `SEQUENCE OF`, the encoding properties are declared after the component names
- The encoding properties are declared at type reference level. If a new type is declared in the ASN.1 grammar based on an existing type reference, then the new type inherits from the base type its encoding properties. The new type can override the inherited properties with its own.

2.1. Auto generation of default ACN grammar

For a given ASN.1 grammar, the Semantix ASN.1 compiler can automatically generate a default ACN grammar with such values in the encoding properties in order to mimic the behavior of uPER. The user can then take the automatically generated ACN grammar and modify some encoding properties according to his/her needs.

For example, for the ASN.1 grammar of Listing 1, the following ACN default grammar is generated:

```
MYMOD DEFINITIONS ::= BEGIN

   MyInt[size 8, adjust -100, encoding pos-int]

   MyInt2[size 10, encoding pos-int]

   MySeq[optionality uper] {
           a1 [size 5, adjust 1, encoding pos-int],
           a2 [size 5, adjust -10, encoding pos-int],
           a3 [size 8, adjust -100, encoding pos-int],
           a4 [size 10, encoding pos-int]
   }

END
```

Listing 3: Automatically generated ACN grammar by the ASN.1 grammar of Listing 1

To automatically generate the ACN grammar, the Semantix ASN.1 compiler is invoked as follows:

```
asn1 -ACND Asn1FileName.asn1
```

3. ACN ENCODING PROPERTIES

3.1. size property

The size encoding property controls the size of the encoding type. It comes in four forms:

3.1.1. Fixed form

This form is used when the size of the encoded type is fixed and known at compile time

Syntax

size *intExpr – the units are provided in the table below*

Example

size 10

size WORDSIZE/2 -- WORDSIZE is an ACN constant defined before

The following table lists the ASN.1 types where the fixed form can be applied as well as the corresponding count unit.

Asn1 Type	Count unit of intExpr
Integer	Bits
Enumerated	Bits
Bit String	Bits
Octet String	Octets
IA5String	Characters
Numeric String	Characters
Sequence/set Of	Elements of sequence/set of

Table 1: ASN.1 types where the size property can be applied

3.1.2. Variable size with embedded length

This form is used when the size of the encoded type is variable. The Asn1/ACN compiler encodes a length field before the actual data of the encoded type which holds the number of the encoded elements that follow. For integer and enumerated types, the length field is always one byte and the count unit is bytes (in order to be consistent with uPER – which means even huge INTEGERs can be supported, up to 255x8bits). For octet strings, bit strings, character strings and sequence/set of types, the length field size is calculated as in uPER $\lceil \log_2(Max - Min) \rceil$ and holds the number of octets, bits, character or sequence/set of elements respectively.

Syntax

size auto

Example

size auto

3.1.3. Variable size with length specified in external field

This form of size property is functionally equivalent with the previous one. The main difference is that the length field is an external field provided in the ACN grammar

Syntax

size *field*

Example

size length	*length* is an integer field defined in the same scope with the encoded type
size header.length	*header* is a sequence type defined in the same scope with the encoded type and which contains an integer type component named *length*

This form of size property can be applied to bit string, octet string, character strings and sequence/set of types.

3.1.4. Null terminated

This form can be applied only to integer or enumerated types and when their encoded property is set to ASCII or BCD (see next section). The terminating character is null (0) for ASCII encodings and the nibble 'F' for BCD encodings.

Syntax

size null-terminated

Example

size null-terminated

3.2. encoding property

The encoding property can be applied only to integer, enumerated and real types.

Syntax

encoding *encvalue*

where *encvalue* is one of pos-int, twos-complement, BCD, ASCII, IEEE754-1985-32 and IEEE754-1985-64

Example

encoding pos-int

encoding BCD

Encoding value	Applicable ASN.1 types	Remarks
pos-int	Integer, enumerated	The ASN.1 integer must have constraints so that only positive values are allowed.

		Otherwise the compiler will report an error.
twos-complement	Integer, enumerated	
ASCII	Integer, enumerated	The ASCII code of the sign symbol ('+' or '-') is encoded first (mandatory) followed by the ASCII codes of the decimal digits of the encoded value. For example, the value 456 will be encoded in the four ASCII codes: 42 (i.e. '+'), 52, 53, 54.
BCD	Integer, enumerated	The ASN.1 integer must have constraints so that only positive values are allowed. Otherwise the compiler will report an error.
IEEE754-1985-32	Real	http://en.wikipedia.org/wiki/IEEE_754-1985
IEEE754-1985-64	Real	(same link as above)

Table 2: ASN.1 properties where the encoding property can be applied

3.3. endianness property

The endianness property can be applied only to fix size integers (and in particular when the size is 16, 32 or 64 bits), enumerated and real types and determines the order of the encoded bytes. For more information please refer to http://en.wikipedia.org/wiki/Endianness

Syntax

endianness *endianness-value*

where *endianness-value* is big or little

Example

endianness little

endianness big (Default)

Encoding value	Applicable ASN.1 types	Remarks
Big	Integer, enumerated, Real	The 32 bit integer value 0xAABBCCDD will be transmitted as follows: 0xAA, 0xBB, 0xCC, 0xDD
Little	Integer, enumerated, Real	The 32 bit integer value 0xAABBCCDD will be transmitted as follows: 0xDD , 0xCC , 0xBB, 0xAA

Table 3: endianness property description

3.4. adjust property

This property can be applied only to integer, enumerated types and to any type with size constraints (sequence/set of, octet strings etc). In integer and enumerated types it affects the integer value itself, while in types with size constraints it affects the encoding of the length determinant. In the encoding process, the (integer) encoded value is the value of the integer encoded type minus the value of the adjust property. In the decoding process the reverse action is performed.

Encoding process:

encodedValue := actualValue – adjust

Decoding Process:

actualValue := decodedValue + adjust

Syntax

adjust *intExpr*

Example

adjust 10

This property has been introduced so that ACN can mimic the uPER behavior. For example, assume the following ASN.1 type

MyInt ::= INTEGER(5..10)

In uPER, this type is encoded as positive integer using 3 bits. Moreover, the actual encoded value is the value of MyInt minus 5. (Otherwise values 8, 9 and 10 would be impossible to encode in three bits).

Hence, ACN properties in order to encode MyInt exactly as in uPER are:

MyInt [size 3, encoding pos-int, adjust 5]

3.5. align-to-next property

This property can be applied to any ASN.1 type, and allows the type to be encoded at the beginning of the next byte or word or double word of the encoded bit stream.

Syntax

align-to-next *alignValue*

Example

align-to-next byte -- 8 bits

align-to-next word – 16 bits

align-to-next dword – 32 bits

3.6. encode-what property

This property can be applied only to enumerated types and controls whether the enumerant values will be encoded or their indexes.

Syntax

encode-what *value*

Example

encode-what values

encode-what indexes

3.7. true-value and false-value properties

These two mutually exclusive properties can be applied only to Boolean types and determine what value will be used to encode TRUE or FALSE values.

Syntax

true-value *bitStringValue*

false-value *bitStringValue*

Example

true-value *'111'B*

false-value *'0'B*

3.8. optionality property

This property can be applied only to sequence or set types and controls how the presence or absence of optional components is encoded. If optionality is set to value 'uper' optional components are handled exactly as in uPER (via a bitmask in the beginning). If optionality is set to 'manual', the presence or absence of optional components is determined by the value of external Boolean fields as explained in the next paragraph (via "present-when").

Syntax

optionality *value*

Example

optionality uper

optionality manual

3.9. present-when property

This property can be applied to any OPTIONAL ASN.1 type provided that the optionality property of the parent sequence or set type is set to 'manual'

Syntax

Present-when *booleanFld*

Example

```
MySeq ::= SEQUENCE {
  alpha       INTEGER,
  beta        BOOLEAN,
  gamma       REAL      OPTIONAL
}
```

Listing 4: Sample ASN.1 grammar

```
Seq[optionality manual] {
  alpha [],
  beta [],
  gamma [present-when beta, encoding IEEE754-1985-64]
}
```

Listing 5: ACN grammar for ASN.1 grammar of Listing 4

In the above example, gamma field is present only when beta is TRUE.

3.10. determinant-uper and determinant-tag properties

These two mutually exclusive properties can be applied only to choice types and determine how the active alternative is encoded. In case of determinant-uper, as the name suggests, the process is the same with uPER (a small positive integer is encoded at the beginning of the choice with index of the active alternative). In case of determinant-tag, the active alternative is determined by an external enumerated field which must have the same names in its enumerants as the names of the choice alternatives.

Syntax

determinant-uper

determinant-tag *enumFld*

Example

```
MySeq ::= SEQUENCE {
  activeColor       ENUMERATED {green, red, blue}
  beta              BOOLEAN,
  colorData         CHOICE {
                          green REAL,
                          red   INTEGER,
                          blue  IA5String (SIZE(1..20))
                    }

}
```

Listing 6: Sample ASN.1 grammar

```
MySeq [] {
  activeColor [],
  beta        [],
  colorData   [determinant-tag activeColor]
}
```

Listing 7: ACN grammar for ASN.1 grammar of Listing 6

In the example above, the active alternative in *colorData* choice is determined by the enumerated field activeColor.

4. ENHANCED OPTIONS

4.1. Fields introduced in the ACN grammar

In some cases, the value for the encoding properties "size", "present-when" and "determinant-tag" may be another field. These fields do not carry semantic (i.e. application specific) information but are used only in the decoding and encoding processes. Therefore these fields may not exist in the ASN.1 grammar but introduced only in the ACN one. For example, Listing 4 can be modified as follows:

```
MySeq ::= SEQUENCE {
  alpha       INTEGER,
  gamma       REAL        OPTIONAL
}
```

Listing 8: The revised ASN.1 grammar of Listing 4. Field 'beta' is missing.

```
Seq[optionality manual] {
  alpha [],
  beta BOOLEAN [],  -- exists only in the ACN file, not the ASN.1 one
  gamma [present-when beta, encoding IEEE754-1985-64]
}
```

Listing 9: Revised ACN grammar for the ASN.1 grammar of Listing 9. Field 'beta' along with the type (BOOLEAN) is introduced.

Please notice that field 'beta' does not exist in the ASN.1 grammar but it was introduced only in the ACN grammar.

4.2. Parameterized encodings and deep field access

There are cases where the length field of a sequence of (or choice determinant, or optionality determinant etc) is not at the same level (i.e. components of a common parent) as the sequence of itself. Actually there are three distinct cases:

a) The length determinant is one or more levels more deeply than the `SEQUENCE OF`
b) The `SEQUENCE OF` is one or more levels more deeply than the length determinant
c) The length determinant and the `SEQUENCE OF` are located in completely different nodes which just have a common ancestor.

These three cases are explained in more detail in the following sub-paragraphs

4.2.1. Length determinant is below current node

This case is illustrated in Figure 1. Field `secondaryHeader`, which is optional, is present when the `secHeaderFlag` in the `primaryHeader` is true.

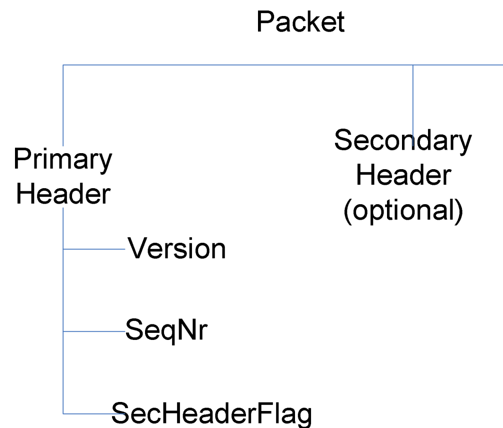

Figure 1. Deep field access – case a.

The corresponding ASN.1 / ACN grammar is:

```
--ASN.1 DEFINITION
  Packet ::= SEQUENCE {
        primaryHeader SEQUENCE {
              version INTEGER,
              seqNr  INTEGER,
              secHeaderFlag      BOOLEAN
        },
        secondaryHeader SEQUENCE {...} OPTIONAL
  }

-- Encodings definition
  Packet  {
        primaryHeader[] {
              version [],
              seqNr [],
              secHeaderFlag []
        }
        secondaryHeader [present-when primaryHeader.secHeaderFlag]
  }
```

Listing 10: ACN grammar demonstrating access to fields at different levels

As shown in the example above, to access a "deep field" located in a child structure we follow the C language notation i.e. fieldname.fieldname.fieldname etc. until we reach the field we want.

4.2.2. Length determinant is above current node

This is the case where the array (sequence of_ is one or more levels more deeply than the length determinant. For example, field "`nrCalls`", which is a top level field, contains the number of calls in the array "`calls`" located under "`SourceData`". Obviously, the "`nrCalls`" field is not accessible from the "`calls`" field. To overcome this issue, we must make the `SourceData` structure parameterized. This case is shown in Figure 2.

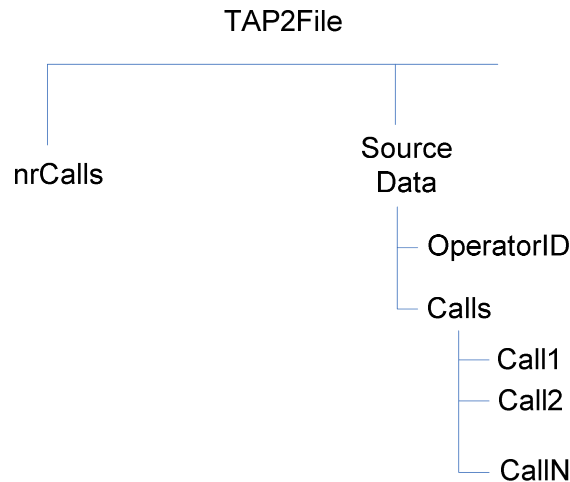

Figure 2. Deep field access – case b.

The corresponding ASN.1 / ACN grammar is:

```
--ASN.1 DEFINITION

TAP2File ::= SEQUENCE {
      nrCalls INTEGER,
      data SourceData
}

SourceData ::= SEQUENCE {
      operatorID IA5String,
      calls SEQUENCE (SIZE(1..100))OF Call
}

--ACN DEFINITION

TAP2File {
        nrCalls [],
        data <nrCalls> [ ] -- nrCalls is passed as a parameter in SourceData
}

SourceData<INTEGER:nElements>
-- nElements is a parameter used in encoding/decoding
-- passed in from the levels above (in this case, TAP2File level)
{
        operatorID [],
        calls[size nElements] -- points to a parameter not a field
}
```

Listing 11: ACN grammar demonstrating parameterized encodings

Please note the "`<>`" in the encoding definition of the SourceData which contains the list with the encoding parameters (in this example, just one).

4.2.3. Length determinant is in completely different subtree

This case is the combination of the two previous cases. A typical case is depicted in Figure 3. In this example, field "`nCalls`", which is located under "`header`" record, contains the number of calls in the "`calls`" array under "`SourceData`".

Field `nCalls` (length determinant) and field `calls` (the `SEQUENCE OF`) are components of two sibling structures (`Header`, `SourceData`) and have no access to each other.

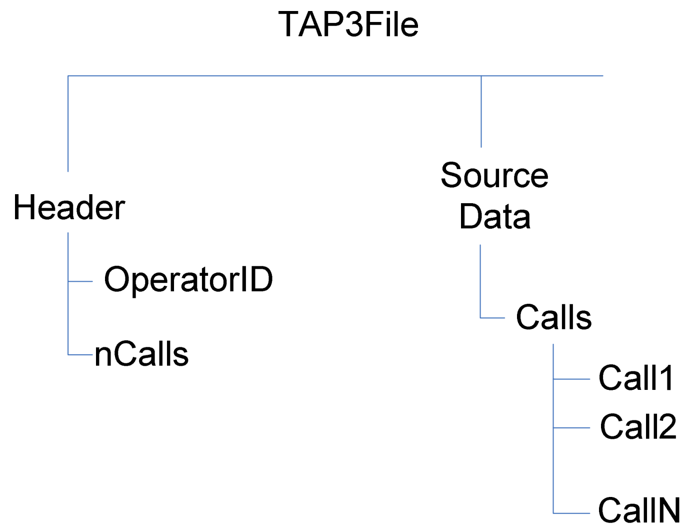

Figure 3. Deep field access – case c.

To handle this case, we must apply the techniques of both previous cases. The corresponding ASN.1 / ACN grammar would be:

```
--ASN.1 DEFINITION

TAP3File ::= SEQUENCE {
      header Header,
      data SourceData
}

Header ::= SEQUENCE {
      operatorID IA5String,
      nCalls INTEGER
}

SourceData ::= SEQUENCE {
   calls SEQUENCE (SIZE(1..100)) OF Call -- length field is contained in the header.nCalls
}

--ACN DEFINITION


TAP3File {
        header [] {},
        data <header.nCalls> [ ] -- header.nCalls is passed as a parameter
                                 -- in SourceData
}

Header[]{
        operatorID[],
        nCalls[]
    }

SourceData<INTEGER:nElements>    -- parameters
{
        calls[size nElements] – "size" points to a parameter, not a field
}
```

Listing 12: ACN grammar demonstrating parameterized encodings and deep field access