PUBLICLY AVAILABLE SPECIFICATION

ISO/PAS 17684

First edition
2003-12-01

Transport information and control systems — In-vehicule navigation systems — ITS message set translator to ASN.1 format definitions

Systèmes de commande et d'information des transports — Systèmes de navigation dans les véhicules — Traducteur de l'ensemble des messages ITS en définitions de format ASN.1

Reference number
ISO/PAS 17684:2003(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of normative document:

— an ISO Publicly Available Specification (ISO/PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50 % of the members of the parent committee casting a vote;

— an ISO Technical Specification (ISO/TS) represents an agreement between the members of a technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/PAS or ISO/TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/PAS or ISO/TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/PAS 17684 was prepared by Technical Committee ISO/TC 204, *Intelligent transport systems*.

Introduction

Working Group 11 of the Technical Committee ISO/TC 204 is responsible for the ITS information concerning navigation and route guidance. This committee, during the preparation of ISO 15075, *Transport information and control systems — In-vehicle navigation systems — Communications message set requirements*, realized that the ISO/IEC directives require a formal description by ASN.1 whenever an information data structure, such as message set, is defined. ASN.1 is a well-defined language with significant descriptive power; however, due to the simplicity of almost all messages used in navigation and route guidance systems, they can be more simply expressed in tabular form. Since it is a common practice among navigation system engineers to define a message set by tables, the main part of this document specifies messages as tables. Faced with the situation of different types of descriptions for a message set, one practical and the other formal, ISO/TC 204 started a project that bridges these descriptions instead of writing down one ASN.1 description for ISO 15075. This document, which allows engineers to write message sets for navigation and route guidance in a tabular form that is automatically translated to ASN.1 description, is a result of that project.

In many navigation systems, traffic and travel information is transferred to and from vehicles and traffic information centers via transmitted messages. To make on-board navigation units compatible with navigation facilities, each message used in the system has to be precisely defined. The internationally accepted Abstract Syntax Notation One (ASN.1) is a powerful tool for describing information data structures. ISO/IEC has agreed that data structures, such as a message set, which define a standard should be formally described in ASN.1. As a consequence, ISO/TC 204 decided that information data structures such as the format of messages including navigation messages must be defined in ASN.1.

Navigation engineers, however, usually use much simpler tabular forms to describe their message sets. As the degree of complexity required in descriptions of navigation message sets does not require the full descriptive power of ASN.1, it was easy to imagine that some subset of ASN.1 could be defined for a simpler description of navigation message sets. Therefore, the first task was to define the subset of ASN.1 most suitable for this purpose. However, instead of directly extracting a subset of ASN.1, an intermediate language, DNF (Descriptor Normal Form; see Clause 5), was proposed. The intermediate DNF language must retain its compatibility with ASN.1, but can be simplified since most of the messages used in navigation are linear in the sense that they are simply written in the lines of a table. Thus, the descriptive power of DNF is strictly limited and any description defined by DNF should also be defined by ASN.1. This means that the set of all possible DNF descriptions is a subset of descriptions available in ASN.1. In this sense, DNF is a subset of ASN.1.

The bridge between the simple DNF tabular descriptions and their respective formal ASN.1 descriptions is a compiler “dnf2asn1”. As shown in the following figure, a DNF description is translated to ASN.1 description by the compiler “dnf2asn1.”

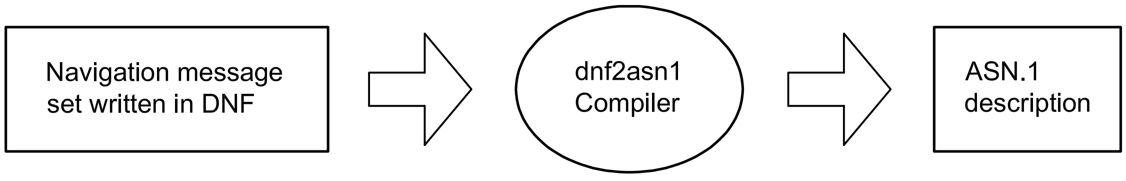


Figure 1 — Function of the bridge compiler dnf2asn1

With this compiling path, any navigation message set that is described in DNF is automatically converted to an ASN.1 description. By inserting an additional processing step prior to DNF, e.g. inserting another language definition and its compiler to DNF, a message set described in the new language can be converted to first DNF by the new compiler and then to ASN.1 by dnf2asn1. In this way, an ASN.1 description of the original message set is generated. If the new language is based on a tabular form, then this double-bridge system can convert a tabular-form description of a message set to its ASN.1 equivalent. It is well known that the tabular form description of message set is much easier for navigation engineers to understand, define and maintain. However, the type of tabular form which best serves this purpose has not yet been clearly defined. Therefore, this document presents an example of a tabular-form message-set description language and its compiler to DNF. Many aspects of the language were kept open for user definition so that descriptive features could be flexibly changed. As this tabular form message set/compiler is an example, it may not accommodate all the needs of a user, and thus it is recommended that a more specific message set/compiler be generated.

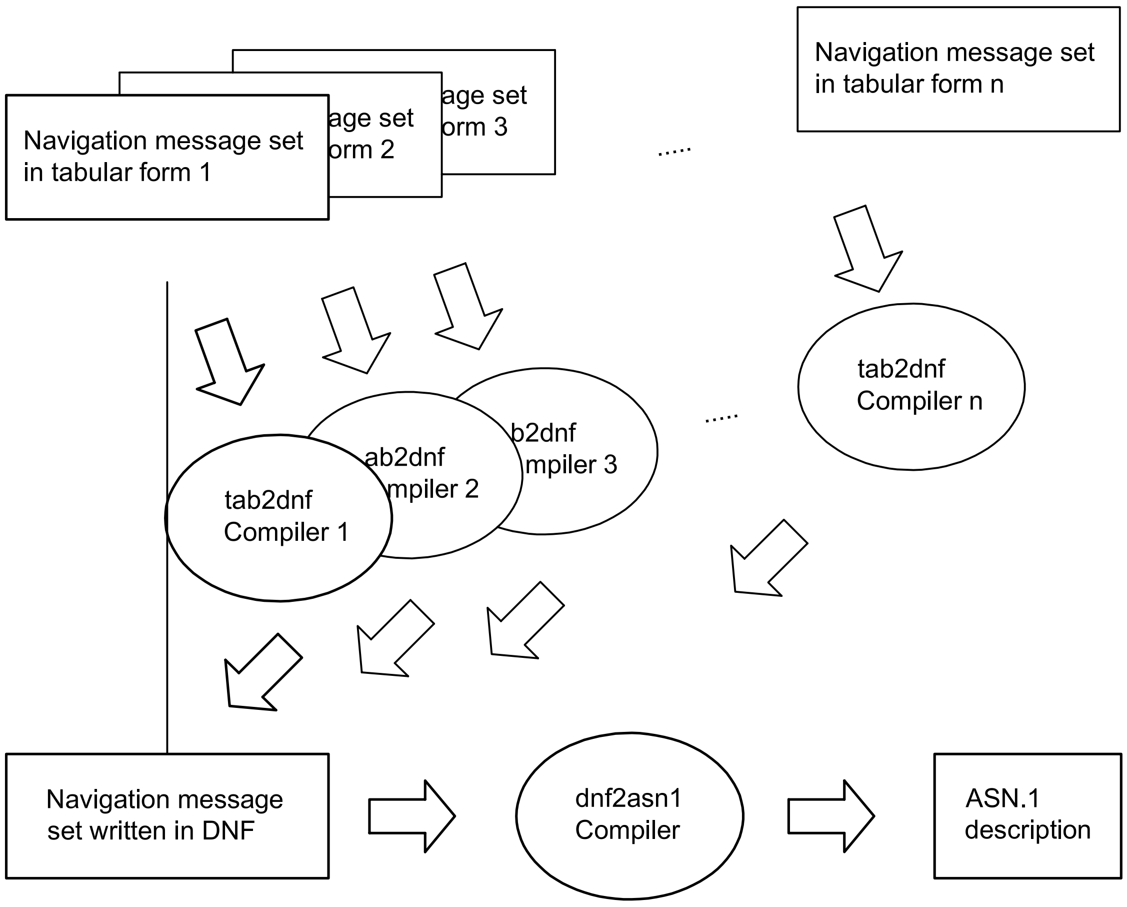


Figure 2 — Conversion of the tabular-form navigation message set into ASN.1

Transport information and control systems — In-vehicule navigation systems — ITS message set translator to ASN.1 format definitions

1 Scope

This Publicly Available Specification specifies a method that can be used to define navigation message sets in tabular form with a subsequent translation into a corresponding ASN.1 description. An intermediate language called Descriptor Normal Form (DNF), which is a subset of ASN.1, is specified and used as an intermediate description between a tabular form and its ASN.1 description.

A tabular-form message-set description language called Message Set Tabular Form (MSTF) is included as an example of a tabular form definition.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 8824-1:1998, *Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation — Part 1*

ISO/IEC 10646-1:2000, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply. The following terms are defined in ISO/IEC 8824-1:1998 specification, bit string type, boolean type, character, character string types, choice types, component type, encoding, (ASN.1) encoding rules, enumerated types, false, integer type, module, null type, octetstring type, real type, recursive definitions, selection types, sequence types, sequence-of types, spacing character, simple type, tag, true, type, type reference name, value, value set, white-space.

3.1
DNF character set
set of characters used in the DNF notation

NOTE See 5.4.

3.2
floating-point number
number capable of being specified by the formula $M \times 10E$ where M and E are two integers called the mantissa and the exponent, respectively

3.3
floating-point type
type consisting of all possible floating-point numbers plus the two values +∞ and -∞

NOTE The floating-point type is a subset of the real type in ASN.1. The real type accommodates bases other than 10.

3.4
items
named sequences of characters from the DNF character set which are used to form the DNF notation

NOTE See 5.4.

4 Message Set Tabular Form description

4.1 General

Message Set Tabular Form (MSTF) describes a message set by message components. A message component is defined by a line or lines, where a line consists of a sequence of boxes or cells. Each cell contains either a string of characters or is empty. A string of characters contained in a cell may be one of the following component items:

— a reserved word;

— a data type;

— a data item;

— a special indicator composed of one or two special characters;

— a comment denoted by the special indicator "//"at the beginning of the comment character string.

Some of the data items used in MSTF are DNF identifiers (see 5.5.2). These component items are arranged in cells to form a line in accordance with the rules for MSTF line format, and then the lines are structured to form a message component in MSTF. Major message components are structured in a block of lines where a block is separated by a line of empty cells.

The first message component of a message set is the TITLE component.

4.2 MSTF description

4.2.1 Character set

The characters that can appear in MSTF are the same characters used in DNF (see 5.4.1).

4.2.2 Special indicators

The following indicators, specified by one or two special character(s) and separated by commas, are used as special indicators: *,||,{},[],//

4.2.3 Reserved words

The following words are reserved in MSTF description.

— EXPORT

— IMPORT

— TITLE

— IF

— ELSEIF

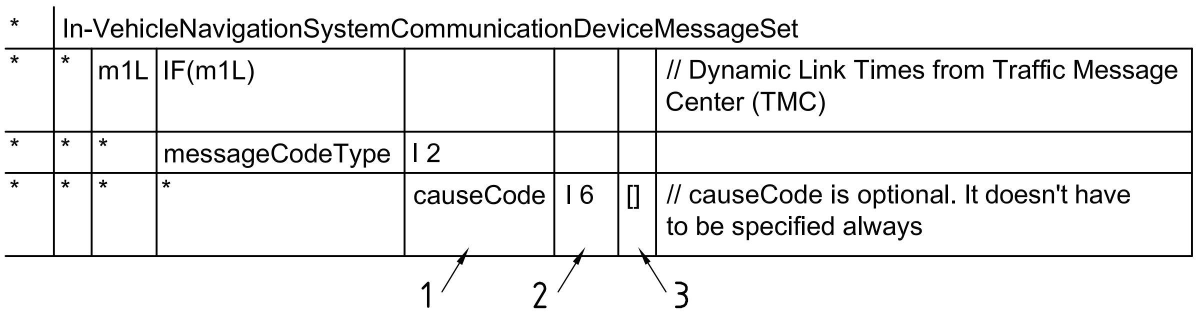
4.2.4 General line format

Each MSTF line is a sequence of cells organized in a *Label Value Control* (*LVC)* format. An *LVC form* is composed of a sequence of cells with the following items:

a) Label — enumerated type label name or enumerated type name (see 5.5.5);

b) Value — identifier, integer value (see 5.6.1.d) or range of values for the label;

c) Control —special indicator or switch indicator specifying a control applied to the label or type.



*	In-VehicleNavigationSystemCommunicationDeviceMessageSet						
*	*	m1L	IF(m1L)				// Dynamic Link Times from Traffic Message Center (TMC)
*	*	*	messageCodeType	I 2			
*	*	*	*	causeCode	I 6	[]	// causeCode is optional. It doesn't have to be specified always

Key

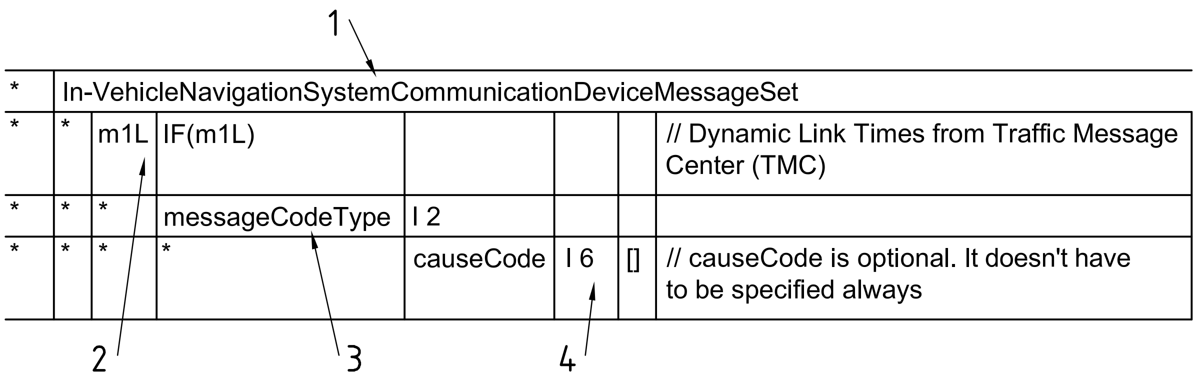
1 label

2 value

3 control

Figure 3 — A line with LVC form

When these three items are present in a line, they shall be in this (L V C) order. Every non-empty line shall have its “Label” item, but the “Value” and “Control” items are optional. Thus, there are four different combinations of the cell items: L, LC, LC or LVC depending on the number and type of items.



*	In-VehicleNavigationSystemCommunicationDeviceMessageSet						
*	*	m1L	IF(m1L)				// Dynamic Link Times from Traffic Message Center (TMC)
*	*	*	messageCodeType	I 2			
*	*	*	*	causeCode	I 6	[]	// causeCode is optional. It doesn't have to be specified always

Key

1 L Type — The line is defined only by the label item

2 LC Type — The line is defined by the label item and the control item

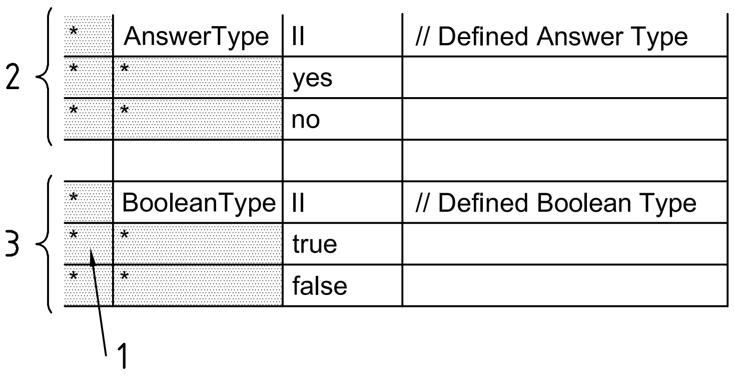
3 LV Type — The line is defined by the label item and the value item

4 LVC Type — The line is controlled by the label item, the value item, and the control item

Figure 4 — The four different line forms

4.2.5 Message structure

A message component is a line or a block of lines. When a message component is a block of lines, the block is delimited by an empty line and structured by a special nesting indicator (default character is *, but may be changed), or indicators placed in the left-most cell or cells of each line. If the number of nesting indicators on a line is N then the line is nested to belong to the line which has N-1 nesting indicators and is above and nearest to the line.



*	AnswerType	II	// Defined Answer Type
*	*	yes	
*	*	no	
*	BooleanType	II	// Defined Boolean Type
*	*	true	
*	*	false	

Key

1 message structure

2 “yes” and “no” lines nested together with the “Answer Type” line

3 “true” and “false” lines nested together with “Boolean Type” line

Figure 5 — Message component blocks

4.2.6 Message components

A message contains the following components.

a) Label definition

A label is assigned to each message by name. The simplest message components have an identifier with the same name as its label:

EXAMPLE simple label

*	Label	identifier

b) Sequence components

A sequence component is defined by a label and the data-item lines that follow, nested according to the label definition. A sequence component signifies that all data items defined in the component will appear in the same order as defined.

EXAMPLE sequence component

*	Label	
*	*	Data Item1
*	*	Data Item2
*	*	Data Item3

A switched sequence of components is described by switch indicators incorporating the reserved words IF, ELSEIF, or ELSE.

EXAMPLE switched sequence of components

	Label A	IF (Case Value)
	*	Data ItemA1
	*	Data ItemA2
	*	Data ItemA3
	Label B	ELSEIF (Case Value)
	*	Data ItemB1
	*	Data ItemB2
	*	Data ItemB3
	Label C	ELSE
	*	Data ItemC1
	*	Data ItemC2
	*	Data ItemC3

Depending on the Case Values, the sequence of data items varies from A1,A2,A3 to B1,B2,B3 to C1,C2,C3.

c) Optional components

An optional component is defined by a label and the special indicator “[]” (square brackets).

EXAMPLE optional component

*	Label	identifier		

	Label			
		Data Item1	I6	
		Data Item2	A	[]

d) Repeat component

A repeat component is defined by a label and the special indicator “{}” (curly brackets).

EXAMPLE repeat component

*	Label	{}		

	Label			
		Data Item1	I6	
		Data Item2	A	{}

e) Choice component

A choice component is defined by a label and the special indicator “||” (vertical bars). The data-item lines that follow the label line are nested according to the label definition. A choice component signifies that one or more of the data items defined in the component will appear in the message component without repetition.

EXAMPLE choice component

*	Label	||
*	*	Data Item1
*	*	Data Item2
*	*	Data Item3

f) Reference component

A reference component signifies the outside reference to a message set defined externally. It is specified by the reserved words IMPORT or EXPORT.

EXAMPLE reference component

IMPORT	Other Message Set Components
EXPORT	Other Message Set Components

g) Title component

A title component defines the message set module name. It is specified by the reserved word TITLE.

EXAMPLE title component

TITLE	This Message Set Components

h) Comment

Anything on a line after a double slash ("//") shall be treated as a comment but ignored by the compiler. It can be placed anywhere except at the head of an LVC form or in a composite element.

EXAMPLE comment component

*	In VehicleNavigationSystemCommunicationDeviceMessageSet						
*	*	M1L	IF(m1L)				// Dynamic Link Times from Traffic Message Center (TMC)
// m1L Detail							
*	*	*	messageCodeType	I 2			
*	*	*	*	causeCode	1 6	[]	// causeCode is optional. It doesn't always have to be specified.

4.2.7 Data types

The MSTF data types correspond to the ASN.1 data types. The former are listed in the following tables. (Refer to Clause 5 for the detailed explanation of each data type.)

a) Simple data types

MSTF Types	ASN.1 Types	Limitation
Bool	BOOLEAN	
I	INTEGER	Named Numbers not allowed
enumerated	ENUMERATED	
F	REAL	Only base 10, but standard decimal notation is allowed in DNF.
Bit	BIT STRING	Named Bit List not allowed, only 'B notation allowed for bit string constants.
O	OCTET STRING	Only 'H notation allowed for octet string constants.
NULL	NULL	Only allowed in choice types
A.	CHARACTER STRING	All DNF strings are UTFSString types in ASN.1. ASN.1 provides a dozen other choices for restricted character strings, and allows unrestricted character strings.

b) Subtype elements

MSTF Types	ASN.1 Types	Limitation
value	Single value	Constants that appear in DNF are translated into a single value type.
Low .. High	Value range	Only I for F types
size	Size constraint	Only Bit or O or A types (Fixed size)
Min. max.	Size constraint	Only Bit or O or A types (Fixed size)

5 Specification of Descriptor Normal Form (DNF)

5.1 General

This clause specifies a notation, called Descriptor Normal Form (DNF), for abstract syntax notation.

This clause defines a number of simple types and specifies a method for referencing these types, for limiting the range of these types, and for writing values of these types.

This clause defines mechanisms for constructing more complex types from more basic types, and specifies a notation for defining such complex types.

This clause defines each of these simple types and more complex types in terms of types defined in ASN.1, and explains how to translate a valid DNF description into a valid ASN.1 description.

5.2 Overview of DNF

DNF allows the specification of the structure of a set of data components. At the lowest level is a set of simple types, which are built-in. The simple types allowed in DNF are a subset of the built-in types allowed in ASN.1. In particular, they include integers, floating-point numbers, booleans, strings of characters, strings of bits, and strings of octets. DNF also allows the definition of enumerated types, where a type specifies a set of identifiers as its values.

EXAMPLE The enumerated type TrafficLightColors could be defined as the set of values "red", "green", and "yellow".

There are also rules for creating more complex types out of simpler types. The following represent three of the more complex types allowed in DNF:

a) concatenating a sequence of component types;

b) choosing from among one or more component types;

c) repeating a component type 0 or more times.

By nesting these methods a quite complex data structure specification can be created.

Finally, DNF allows the user to give a name to a component type, and to use that name in other type definitions. This allows a structure that appears in many places to be defined only once. These type definitions may, in fact, be recursive, as they can be in ASN.1.

5.3 DNF Notation

5.3.1 Introduction to notation

The DNF notation consists of a sequence of characters from the DNF character set specified in 5.4.

Each use of the DNF notation contains character sets grouped into items. All sequences of characters forming DNF items and the names of each are specified in 5.5.

The DNF notation is specified in 5.6 and following subclauses. These subclauses specify the collection of sequences of items which form valid examples of the DNF notation and the semantics of each sequence.

The formal notation used to specify these collections is defined in the following subclauses.

5.3.2 Extended Backus-Nauer Form (EBNF)

DNF is precisely and unambiguously described below using a meta-language, e.g. a language used to talk about another language. The meta-language used is a version of Extended Backus-Nauer Form (EBNF). EBNF, as popularized by Wirth[1)] in his description of the language Pascal, is one standard way to describe languages like DNF.

5.3.3 Description of EBNF

5.3.3.1 Productions in EBNF

EBNF defines a language by giving a list of productions. Each production consists of a non-terminal symbol on the left of an "::=" symbol, with a sequence of non-terminals, terminals, and special symbols on the right side. The special symbols define how the terminals and non-terminals can be combined in the language.

1) Jensen, Kathleen, and Niklaus Wirth, PASCAL User Manual and Report, Second Edition, Springer Verlag, Berlin, ISBN 0-387-90144-2, 1974.

When describing EBNF in this document, the word "component" will be used to indicate a terminal, a non-terminal, or a combination of terminals and other components that indicates a choice, a sequence, an optional item, or a repetition.

5.3.3.2 Layout

Blank lines separate productions. A production may extend for a number of lines, but ends when a blank line is encountered.

5.3.3.3 Non-terminals.

Non-terminals begin with an open angle bracket "<". This is followed by one or more letters or digits, the first of which shall be a letter. The non-terminal ends with a closing angle bracket ">".

EXAMPLE <a>, <identifier2>, and <typeIdentifier> are all valid non-terminals.

Non-terminals are placeholders for things that are specified in productions. They must be expanded (replaced by things on the right-hand side of a production) to complete the specification.

5.3.3.4 Terminals

Terminals are strings of characters that appear in the final language. Any characters or character sequences that are described below in 5.5 are valid terminals. They usually are written just as they will appear in the final language. However, confusion can arise because most of the special symbols that are used by the EBNF meta-language are also valid terminals in DNF. Therefore the following seven individual characters appear between double-quote characters ("..") when they are intended as terminal characters in DNF.

EXAMPLE | [ ] { } < >

In addition, the string "::=" will appear between double-quote characters when it indicates a terminal string.

5.3.3.5 Sequences of components

Components following one another in sequence are written one after another on the right side of a production.

5.3.3.6 Optional components

A component enclosed between square brackets [ ] is optional. Either 0 or 1 copy of the component shall be included.

EXAMPLE a [b] c indicates either the sequence ac or the sequence abc.

5.3.3.7 Repeated components

A component enclosed in curly brackets { } can be repeated 0 or more times.

EXAMPLE { a } indicates either an empty sequence, a, aa, aaa, etc.

5.3.3.8 Choice among components

A sequence of components separated by the vertical bar character "|" indicates that exactly one of the components is to be included.

EXAMPLE 1 x | y | z indicates one of x or y or z.

EXAMPLE 2 The production
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
indicates that the non-terminal <digit> represents a single base-10 digit.

5.3.3.9 Precedence rule

The sequence operation has higher precedence than the choice operation.

EXAMPLE The EBNF expression

<comp1> <comp2> | <comp3> <comp4>

means to chose one of the two-component sequences <comp1> <comp2> or <comp3> <comp4>, as opposed to a three component sequence starting with <comp1> and ending with <comp4> with either <comp2> or <comp3> in the middle.

5.3.3.10 Using EBNF

The EBNF notation for the entire DNF language will appear in 5.10. The non-terminal <messageSetSpecification> on the left side of the "::=" in the first production is the start symbol for the language. Starting with this start symbol and successively replacing non-terminals by their definitions until all non-terminals are replaced leads to a valid specification in DNF.

5.3.3.11 Recursion

Rules in EBNF can be recursive. In this case the productions are to be continuously reapplied until no new sequences are generated.

NOTE In many cases, such reapplication results in an unbounded collection of allowed sequences, some of which may themselves be unbounded. This is not an error.

5.4 DNF character set

5.4.1 Characters in DNF items

The characters that can appear in a DNF item are the following, except as specified in 5.4.3 and 5.4.4 below.

— a to z

— A to Z

— 0 to 9

— : = , { } < > . ( ) [ ] - ' " / * ;

5.4.2 EBNF notation for selected sets of characters

The following definitions shall be used in this document for the various sets of characters.

— <lowerCaseLetter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

— <upperCaseLetter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

— <letter> ::= <upperCaseLetter> | <lowerCaseLetter>

— <nonZeroDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

— <digit> ::= 0 | <nonZeroDigit>

— <letterOrDigit> ::= <letter> | <digit>

— <upperLetterOrDigit> ::= <upperCaseLetter> | <digit>

5.4.3 Characters in string values

Where the notation is used to specify the value of a character string type, all graphic symbols for the character set may appear in the DNF notation, surrounded by double-quote characters (").

5.4.4 Characters in comments

Additional (arbitrary) graphic symbols may appear in the comment item (see 5.5.10).

5.4.5 Upper and lower case

Upper- and lower-case letters shall be regarded as distinct.

5.5 DNF items

5.5.1 General rules

5.5.1.1 The following subclauses specify the characters in ASN.1 items. In each case, the name of the item is given, along with the definition of the character sequences which form the item.

5.5.1.2 Each item specified in the following subclauses shall appear on a single line, and (except for comment and alphanumeric string value items) shall not contain white-space.

5.5.1.3 The length of a line is not restricted.

NOTE For practical reasons, some DNF compilers may put an upper limit on the length of a single line (e.g. 1 024 characters).

5.5.1.4 The items in the sequences specified by this chapter may appear on one line or may appear on several lines, and may be separated by white-space, empty lines, or comments.

NOTE Judicious use of white-space and empty lines can make a specification much easier to read. Annexes A and C demonstrate one way of using indentation to achieve this end. Note that the closing symbol of each structure (")", "}", "]", ">") appears either on the same line with, or lined up vertically with, the corresponding opening symbol ("(", "{", "[", "<"). Each additional level of nesting leads to an additional level of indentation. This sort of formatting is recommended, but it is not required by the language definition.

5.5.1.5 An item shall be separated from a following item by white-space, newline or comment, if the initial character or characters of the following item are permitted for inclusion as the next character(s) in the earlier item.

5.5.2 Identifiers

An identifier shall consist of one or more letters, digits and hyphens. The initial character shall be a lower-case letter. A hyphen shall not be the last character. A hyphen shall not be followed immediately by another hyphen.

In EBNF notation, an identifier is named <identifier> and defined as follows.

```
    <identifier> ::= <lowerCaseLetter> { [ - ] <letterOrDigit> }
```

NOTE This definition is the same as in ASN.1. The hyphen rules prevent confusion between identifiers and comments, and between identifiers and following numbers.

EXAMPLE "a-5" is a single identifier, not "a" followed by "-5" or "a-" followed by "5".

5.5.3 Labels

A label shall consist of the same sequence of characters specified for <identifier>. The two are distinguished by context.

In EBNF notation, a label is named <label> and defined as follows.

```
<label> ::= <identifier>
```

5.5.4 Type names

A type name shall consist of one or more letters, digits and hyphens. The initial character shall be an upper-case letter. A hyphen shall not be the last character. A hyphen shall not be followed immediately by another hyphen.

In EBNF notation, a type name is named <typeName> and defined as follows.

```
<typeName> ::= <upperCaseLetter> { [ - ] <letterOrDigit> }
```

A <typeName> shall not be one of the reserved sequences listed in 5.5.9.

A specific <typeName> shall appear at most once as either a <moduleName> (see 5.5.5), as a <holeIdentifier> (see 5.5.6), as an <enumTypeName> on the left side of the "=" in an enumerated type definition (see 5.7), or as the <definedTypeName> on the left side of the "::=" in a defined type definition (see 5.9).

5.5.5 Module names, enumerated type names, defined type names

A module name, an enumerated type name, and a specification description name shall all consist of the same sequence of characters specified for <typeName>. The different types are distinguished by context.

In EBNF notation, a module name is named <moduleName> and defined as follows.

```
<moduleName> ::= <typeName>
```

An enumerated type name is named <enumTypeName> and defined as follows.

```
<enumTypeName> ::= <typeName>
```

A defined type name is named <definedTypeName> and defined as follows.

```
< definedTypeName> ::= <typeName>
```

5.5.6 Hole identifiers

A hole identifier shall consist of one or more upper-case letters, digits and hyphens. The initial character shall be an upper-case letter. A hyphen shall not be the last character. A hyphen shall not be followed immediately by another hyphen.

In EBNF notation, a hole identifier is named <holeIdentifier> and defined as follows.

```
<holeIdentifier> ::= <upperCaseLetter>
{ [ - ] <upperLetterOrDigit> }
```

A <holeIdentifier> shall not be one of the reserved sequences listed in subclause 5.5.9.

Each <holeIdentifier> shall be unique, and a <holeIdentifier> shall not be the same as a <typeName> used in the specification.

5.5.7 Simple type names

The following character sequences are the names of simple types and are described in 5.6.

I

A

Bit

O

Bool

F

Hole

5.5.8 Simple type values

The values of the simple types described in 5.6 are DNF items.

5.5.9 Reserved words

Type names shall not conflict with the names of simple types given in 5.5.7, the word UFT8String, nor with the ASN.1 reserved words specified in 9.16 of ISO/IEC 8824-1:1995 (E). These ASN.1 reserved words are included as Annex B of this document for reference.

5.5.10 Comments

Anything on a line after a double hyphen ("--") or a double slash ("//") shall be treated as a comment to be read by humans but ignored by the language. It is treated as if the entire comment and terminating newline were a single newline.

Anything between the symbols "/*" and "*/" shall be treated as a comment and shall be equivalent to a single space. Note that these symbols may appear on the same line (perhaps with text preceding the "/*" and following the "*/") or may run across a number of lines.

If character sequences "/*", "//", and "--" appear within string values or within comments, they do not serve to begin comments. If "*/"appears within a comment begun by "//" or "--", it is ignored and does not end that comment.

5.5.11 Single and multiple-character items

The following special characters and special character sequences are DNF items. The name of the symbol or symbol sequence follows in parentheses.

::=	(assignment)
..	(range separator)
=	(equal sign)
(	(opening parenthesis)
)	(closing parenthesis)
{	(opening curly bracket)
}	(closing curly bracket)

<	(opening angle bracket)
>	(closing angle bracket)
[	(opening square bracket)
]	(closing square bracket)
,	(comma)
;	(semicolon)
|	(vertical bar)

5.6 Simple types

5.6.1 The integer type

5.6.1.1 Notation for the integer type: I

An integer type shall be indicated by the notation “I”. This is equivalent to the ASN.1 type INTEGER. It is assumed to be of arbitrary size if not constrained.

NOTE Many ASN.1 compilers limit integers to 32 or 64 bits. If DNF is used to specify a data structure which will be compiled by one of these compilers, it should limit the length of integers.

5.6.1.2 Limiting the range of an integer

If two integers separated by the range separator “..” follow the I, they shall be the lower and upper bounds of the range that the integer can assume. The words MIN and MAX may substitute for lower and upper bounds, respectively, and imply that the range is unbounded below and above, respectively.

EXAMPLE I -50..100 indicates an integer in the range -50 to 100.

5.6.1.3 Specifying the maximum number of bits in an integer

If a single positive integer N follows the I, the integer specified shall be constrained to fit into an N-bit field, and shall lie in the range 0 to (2N - 1).

EXAMPLE I 16 indicates an integer in the range 0 to 216 - 1.

NOTE 1 The field width N for integers may be limited to a fixed range (e.g. 1 to 999) in some DNF compilers. An application that utilizes a larger bound on integers can use unbounded integers. The main reason for this restriction is to prevent the accidental specification of a huge width. For example, if someone intended 65535 (the largest integer expressible in 16 digits) to be the upper bound on a range, but forgot to enter a lower bound, the compiler would understand this to specify an integer with approximately 20 000 digits.

NOTE 2 One reason why a huge integer might be desired is that the integer is not really viewed as an integer, but as a sequence of bits encoding a number of pieces of data. In this case, the Bit or O type should be used instead of I.

5.6.1.4 Specifying an integer value

An integer value is named <intValue> and shall consist of a sequence of digits, possibly preceded by a “-” sign.

```
<nonNegInt> ::= <digit> { <digit> }
<intValue> ::= [ - ] <nonNegInt>
```

NOTE Leading 0s are allowed.

5.6.1.5 Definition of the integer type

The EBNF definition of the integer type is as follows.

```
<intType> ::= I [ <size> | <intValue>..<intValue> ]
```

5.6.2 The alphanumeric character string type

5.6.2.1 Notation for the alphanumeric character string type: A

An alphanumeric character string type shall be indicated by the notation “A”. This is equivalent to the ASN.1 type UTF8String, which is defined in ISO/IEC 10646-1:2000.

NOTE Characters in a UTF8String include not only standard ASCII characters, but also foreign alphabets. The type is designed so that a standard ASCII character is encoded in a single 8-bit octet, while other alphabets require more octets per character.

5.6.2.2 Specifying a fixed-length character string

A positive integer N following the A shall specify a fixed-length string. It signifies that the string shall always consist of exactly N characters.

EXAMPLE A 25 indicates a string with exactly 25 characters.

5.6.2.3 Specifying a range of character string lengths

If two non-negative integers separated by the range separator “..” follow the A, they shall be the lower and upper bounds on the number of characters in the string. The words MIN and MAX shall not replace the integers in this case.

EXAMPLE A 1..10 indicates a string with between 1 and 10 characters.

5.6.2.4 Character string values

An alphanumeric string value shall consist of a string of characters surrounded by double quotes, as “AB Z012 /*”. To represent a double quote in the string, repeat it.

EXAMPLE "AB""C" represents the string AB"C.

The EBNF definition of a character string value is

```
<charStringValue> ::= " { <nonQuoteCharacter> | "" } "
```

where the non-terminal <nonQuoteCharacter> indicates any graphic symbol in the defined character set except for the double-quote character ".

5.6.2.5 Definition of the alphanumeric character string type

The EBNF alphanumeric character string type is named <charStringType>. It and several supplementary non-terminals are defined as follows.

```
<positiveInt> ::= { 0 } <nonZeroDigit> { <digit> }
<size> ::= <positiveInt>
<sizeRange> ::= <nonNegInt>..<nonNegInt>
<charStringType> ::= A [ <size> | <sizeRange> ]
```

5.6.3 The bit string type

5.6.3.1 Notation for the bit string type: Bit

A bit string shall be indicated by the notation “Bit”. This is equivalent to the ASN.1 type BIT STRING. A bit string is a string of 0s and 1s.

5.6.3.2 Specifying a fixed-length bit string

A positive integer N following the Bit shall specify a fixed-length string. It means that the bit string shall always consist of exactly N bits.

EXAMPLE Bit 256 indicates a string with exactly 256 bits.

5.6.3.3 Specifying a range of bit string lengths

If two non-negative integers separated by “..” follow the Bit, they shall be the lower and upper bounds on the number of bits in the bit string. The words MIN and MAX shall not replace the integers in this case.

EXAMPLE Bit 1..100 indicates a string with between 1 and 100 bits.

5.6.3.4 Bit string values

A bit string value shall consist of a string of 0s and 1s surrounded by single quotes and shall end in “B”.

EXAMPLE '0110'B and '111000'B are bit strings.

The EBNF definition of a bit string value is

```
<bitStringValue> ::= ' { 0 | 1 } 'B
```

5.6.3.5 Bit string type

The EBNF definition of the bit string type is as follows.

```
<bitStringType> ::= Bit [ <size> | <sizeRange> ]
```

NOTE <size> and <sizeRange> are defined in 5.6.1.5.6.1.5

5.6.4 The octet string type

5.6.4.1 Notation for the octet string type: O

An octet string shall be indicated by the notation “O”. This is equivalent to the ASN.1 type OCTET STRING. An octet string is a sequence of 8-bit octets.

5.6.4.2 Specifying a fixed-length octet string

A positive integer N following the O shall specify a fixed-length string. It means that the octet string shall always consist of exactly N octets.

EXAMPLE O 256 indicates a string with exactly 256 octets.

5.6.4.3 Specifying a range of octet string lengths

If two non-negative integers separated by “..” follow the O they shall be the lower and upper bounds on the number of octets in the octet string. The words MIN and MAX shall not replace the integers in this case.

EXAMPLE O 1..100 indicates a string with between 1 and 100 octets

5.6.4.4 Octet string values

An octet string value shall be written in hexadecimal, surrounded by single quotes, and shall end in “H”.

EXAMPLE '9AF1E2'H is an octet string value.

NOTE A hexadecimal number is written in base-16 notation with the following 16 digit values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Thus the hexadecimal digit A is equivalent to 10 in base-10 notation, and the hexadecimal digit F is equivalent to 15 in base-10 notation.

The EBNF definition of an octet string value is the following.
<octetStringValue> ::= ' { <digit> | A | B | C | D | E | F } 'H

5.6.4.5 Definition of the octet string type

The EBNF definition of the octet string type is the following.
<octetStringType> ::= O [<size> | <sizeRange>]

NOTE <size> and <sizeRange> are defined in 5.6.2.5.

5.6.5 The boolean type

5.6.5.1 Notation for the boolean type: Bool

A boolean shall be indicated by the notation “Bool”. This is equivalent to the ASN.1 type BOOLEAN. Boolean values are true and false.

The EBNF definition for the boolean type is the following.
<booleanType> ::= Bool

5.6.5.2 Boolean values

The boolean values true and false shall be expressed in DNF as the character sequences TRUE and FALSE. All letters in these character sequences shall be upper-case.

The EBNF definition for a boolean value is the following.
<boolValue> ::= TRUE | FALSE

5.6.6 The floating-point type

5.6.6.1 Notation for the floating-point type: F

A floating-point number shall be indicated by the notation “F”. Floating-point numbers are represented by the ASN.1 type REAL.

5.6.6.2 Limiting the range of a floating-point number

If two floating point numbers separated by “..” follow the F, they shall be the lower and upper bounds of the range that the floating-point number can assume. MIN and MAX may substitute for lower and upper bounds, respectively, and indicate that the number is unbounded above and below, respectively.

EXAMPLE F -3.4159..2.06e27 indicates a floating-point number between -3.14159 and 2.06 x 1 027.

5.6.6.3 Floating-point values

A floating-point value shall consist of an optional “-” sign, a sequence of one or more digits, a decimal point, and a sequence of zero or more digits after the decimal point. This may be followed by an “e” and an integer exponent, indicating that the number before the e shall be multiplied by 10 raised to the power indicated by the exponent.

EXAMPLE 7.85e-25 represents 7.85×10^{-25}, which is equal to 785×10^{-27}

NOTE 12.34, -0.5, 0.0, and 5.e12 are valid floating-point values, but .1, 5, and 0 are not. (A floating-point number shall contain a decimal point and there shall be digits before the decimal point.)

The EBNF definition for a floating-point value is the following.

    <floatValue> ::= <intValue>.[<nonNegInt>] [e <intValue>]

5.6.6.4 Definition of the octet string type

The EBNF definition of the octet string type is the following.

    <floatType> ::= F [<floatValue>..<floatValue>]

5.6.7 The hole type

The EBNF definition for the hole type is the following.

    <holeType> ::= Hole <holeIdentifier>

EXAMPLE Hole NOTDEFINED

The purpose of a Hole type is to serve as a placeholder for a type to be defined later. (For example, defining the specification may be the responsibility of a different group.)

5.7 Enumerated types

Enumerated types define a set of identifiers as the values of a type. The EBNF definition for defining an enumerated type is the following.

    <enumeratedTypeDef> ::= <enumTypeName> = <identifier>
    { “|” <identifier> } ;

EXAMPLE The definition

    DaysOfWeek = monday | tuesday | wednesday | thursday | friday | saturday | sunday;

defines the enumerated type

    DaysOfWeek with values monday, tuesday, wednesday, thursday, friday, saturday, and sunday.

5.8 Complex types

5.8.1 Building types from simpler components

This subclause defines ways to build up more complex types by combining simpler components in various ways. In particular, we have sequence types, choice types and repetition types.

Each of these three types is defined in terms of simpler components. The following EBNF definitions will define a component, which is named <comp>.

a) <comp> ::= <type> | <value> | <sequenceType> | <choiceType> | <repetitionType>

b) <type> ::= <simpleType> | <enumTypeName> | <definedTypeName>

c) <simpleType> ::= <intType> | <charStringType> | <bitStringType> | <octetStringType> | <booleanType> | floatType> | <holeType>

NOTE These definitions lead to indirect recursion, because choice, sequence and repetition types are defined in terms of <comp>, which in turn is defined in terms of choice, sequence and repetition types.

5.8.2 Sequence Types

5.8.2.1 Notation for the sequence type

A sequence of components is named <sequenceType> and defined in EBNF as follows.

    <sequenceType> ::= (<labeledSeqComp>

{ , <labeledSeqComp> })

That is, a sequence shall consist of one or more <labeledSeqComp> separated by commas and enclosed in parentheses. The simplest form of a labeled sequence component is a pair.
<label> <comp>

Each <label> within a single sequence shall be unique. However, duplicate labels may appear in different sequences or in choices, including in components nested within the sequence. The labels are used to name components of the data.

A sequence may have a single component, but this is unusual.

EXAMPLE 1 The following is a valid sequence.
(firstName A 1..15, middleInitial A 1, lastName A 1..30)

EXAMPLE 2 Sequences can be nested.
(name (firstName A 1..15, middleInitial A 1, lastName A 1..30),
street A 1..45,
city A 1..30,
state A 2,
zip I 5)

EXAMPLE 3 Labels can repeat within nested components.
(first (first F, second I), second (first Bit 1..100, second Bool)

5.8.2.2 Optional components in a sequence

Sequences may have one or more optional components. An optional component shall be enclosed in square brackets:
"[" <label> <comp> "]"

EXAMPLE 1 The notation
(first 10, [second "ab"] , third 3.5)
indicates either the ordered triple (first 10, second "ab", third 3.5) or the ordered pair (first 10, third 3.5).

EXAMPLE 2 If every component in a 3-component sequence is optional then there are 8 valid possibilities for the choice of components, including the empty sequence. (Each component can be included or not independently.)

5.8.2.3 Default values for optional components

It is also possible to provide default values in sequences for simple and enumerated types. If a component in a sequence is omitted from the transmitted data, the default value is used for that component. .Tthis concept is encoded by extending the notation for an optional component.
"[" <label> <type> = <value> "]"

The <value> item shall be of the same type as <type>. Leaving this component out of the transmitted data is identical to including it with "value" as the value.

5.8.2.4 EBNF definition of <labeledSeqComp>

Combining the three possibilities for <labeledSeqComp> (required, optional, optional with default value) results in the following definition.
<labeledSeqComp> ::= <label> <comp> |
"[" <label> <comp> "]" |
"[" <label> <type> = <value> "]"

5.8.3 Choice types

A choice from among one or more components is named <choiceType> and defined in EBNF as follows.
<choiceType> ::= "<" <label> <choiceComp>

```
      { "|" <label> <choiceComp> } ">"
      <choiceComp> ::= <comp> | NULL
```

Thus, a choice shall consist of one or more labeled components separated by vertical-bar characters and enclosed in angle brackets < >. Exactly one of the components will appear in the data structure. It is possible to have a choice with a single component, but this is unusual.

It is possible to specify the value NULL as the component part of one (or more) of the choices. Selecting NULL is the same as choosing none of the components in the list.

Each <label> within a single choice shall be unique. However, duplicate labels may appear in other choices or in sequences, including in components nested within the choice. The labels are used to name components of the data.

5.8.4 Repetition types

The repetition type <repetitionType>, used to indicate that an component may be repeated 0 or more times, is defined as follows.

```
      <repetitionType> ::= "{" <comp> "}"
```

NOTE A label shall not precede the component in a repetition type.

EXAMPLE The notation

```
      { I }
```

indicates a sequence of integers of any length (including 0).

5.9 Defined types

Defined types provide a way to give a name to a component and use that name as part of the description of other components. This is especially useful when the same structure appears in a number of places. It can be defined once, given a name, and that name can be used in each of the places where the structure appears. If the structure is later changed, it is necessary to change only the definition of the defined type. This requires one change rather than many changes. By picking descriptive names for defined type names, it is often easier to make a description more understandable.

Defined types can be recursive. The defined type name can appear within the definition of the defined type, or two or more type definitions can mutually refer to one another.

The EBNF definition of a defined type is the following.

```
<typeDefinition> ::= <definedTypeName> "::=" <comp> ;
      <definedTypeName> ::= <typeName>
```

EXAMPLE Defined types can be used to define subpieces of more complex structures.

```
      NameType ::=       ( firstName A 1..15,
                         middleInitial A 1,
                         lastName A 1..30 );
      AddressType ::=    ( street A 1..45,
                         city A 1..30,
                         state A 2,
                         zip I 5 );
      PersonRecord ::=   ( name NameType,
                         address AddressType );
```

5.10 Structure of a DNF specification

A specification description is defined as the following.

```
<specificationDescription> ::= <moduleName> <typeList>
      <typeList> ::= { <enumeratedTypeDef> | <typeDefinition> }
```

The <moduleName> is the name that will be given to the ASN.1 module. There may be zero or more enumerated types defined, interspersed with zero or more type definitions.

DNF requires that any enumerated type be defined before any of the identifiers that are its values are used as part of the definition of a component.

5.11 Summary of the EBNF productions for DNF

<specificationDescription> ::= <moduleName> <typeList>

<moduleName> ::= <typeName>

<typeList> ::= { <enumeratedTypeDef> | <typeDefinition> }

<typeDefinition> ::= <definedTypeName> "::=" <comp>;

<enumeratedTypeDef> ::= <enumTypeName> = <identifier> { "|" <identifier> };

<enumTypeName> ::= <typeName>

<definedTypeName> ::= <typeName>

<identifier> ::= <lowerCaseLetter> { [-] <letterOrDigit> }

<typeName> ::= <upperCaseLetter> { [-] <letterOrDigit> }

<holeIdentifier> ::= <upperCaseLetter>

{ [-] <upperLetterOrDigit> }

<letterOrDigit> ::= <letter> | <digit>

<letter> ::= <upperCaseLetter> | <lowerCaseLetter>

<upperLetterOrDigit> ::= <upperCaseLetter> | <digit>

<nonZeroDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit> ::= 0 | <nonZeroDigit>

<upperCaseLetter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lowerCaseLetter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<comp> ::= <type> | <value> | <sequenceType> | <choiceType> | <repetitionType>

<type> ::= <simpleType> | <enumTypeName> | <definedTypeName>

<simpleType> ::= <intType> | <charStringType> | <bitStringType> | <octetStringType> | <booleanType> | <floatType> | <holeType>

<intType> ::= I [<size> | <intValue>..<intValue>]

<charStringType> ::= A [<size> | <sizeRange>]

<bitStringType> ::= Bit [<size> | <sizeRange>]

<octetStringType> ::= O [<size> | <sizeRange>]

<booleanType> ::= Bool

<floatType> ::= F [<floatValue>..<floatValue>]

<holeType> ::=   Hole <holeIdentifier>

<sizeRange> ::= <nonNegInt>..<nonNegInt>

<value> ::= <intValue> | <charStringValue> | <bitStringValue> | <octetStringValue> | <boolValue> | <floatValue> | <identifier>

<intValue> ::= [-] <nonNegInt>

<nonNegInt> ::= <digit> { <digit> }

```
<positiveInt> ::= { 0 } <nonZeroDigit> { <digit> }
<size> ::= <positiveInt>
<floatValue> ::= <intValue>.[ <nonNegInt> ] [ e <intValue> ]
<charStringValue> ::= " { <nonQuoteCharacter> | "" } "
<bitStringValue> ::= ' { 0 | 1 } 'B;<octetStringValue> ::= ' { <digit> | A | B | C | D | E | F } 'H
<boolValue> ::= TRUE | FALSE;<sequenceType> ::= ( <labeledSeqComp>
{ , <labeledSeqComp> } )
<labeledSeqComp> ::= <label> <comp> |
"[" <label> <comp> "]" |
"[" <label> <type> = <value> "]"
<choiceType> ::= "<" <label> <choiceComp>
{ "|" <label> <choiceComp> } ">"
<label> ::= <identifier>
<choiceComp> ::= <comp> | NULL
<repetitionType> ::= "{" <comp> "}"
```

The non-terminal <nonQuoteCharacter> indicates any graphic symbol in the defined character set except for the double-quote character ".

5.12 Translation of DNF to ASN.1

5.12.1 Translating simple types

Simple types (except for Hole) shall be translated to the corresponding ASN.1 built-in types as specified in 5.6. Restrictions on range and length shall translate to ASN.1 restrictions on range and length. Values that appear as individual components shall map into the ASN.1 simple or enumerated types constrained to that single value.

5.12.2 Translating sequence types

Sequence types shall be translated into ASN.1 SEQUENCE types. Optional components shall be translated into equivalent ASN.1 components with the keyword OPTIONAL appended. Default values shall be translated into equivalent ASN.1 components with the keyword DEFAULT and a value appended.

5.12.3 Translating choice types

Choice types shall be translated into ASN.1 CHOICE types.

5.12.4 Translating repetition types

Repetition structures shall be translated into ASN.1 SEQUENCE OF types.

5.12.5 Handling labels in translations

The labels on the various components in sequence and choice structures shall be used as tag names in the ASN.1 encoding.

5.12.6 Making types extensible

All of the CHOICE and SEQUENCE structures in ASN.1 should have "..." as their last component. This makes the structure extensible. If more components are added to the end of the CHOICE or SEQUENCE, then programs using the old specification will still be able to process data encoded by the new specification. As long as none of the newly defined components is included, everything works normally. If a component that the receiving program does not recognize appears, it will not be able to understand it; but it will be able to recognize it as something added via an extension rather than as an error.

5.12.7 Translating the Hole type

There are many possible ways of translating the Hole type, some of which are rather complex. A simple way to translate the Hole type is by putting the value given to the <holeIdentifier> (for example TYPENAME) in place of the Hole in the ASN.1 translation. Then, at the end of the specification define TYPENAME via the following production

```
TYPENAME ::= SEQUENCE { ... }
```

This defines TYPENAME to be an extensible empty sequence. Therefore, the user has the option of adding ASN.1 components to the end of this sequence (if he or she wants programs compiled using the old definition to continue working) or the option of replacing the type entirely. Of course, the simplest option is to re-compile the DNF definition once the type is known.

For this method of handling Holes, it would not be necessary for a <holeIdentifier> not to have any lower-case letters, but some alternate approaches require that there be no lower-case letters. Therefore, this document requires that a <holeIdentifier> shall have no lower-case letters.

5.12.8 Translating enumerated types

Enumerated types translate into ASN.1 enumerated types.

5.12.9 Translating defined types

Defined types shall be translated into a production with the <definedTypeName> as the type name on the left side of the assignment operator.

5.12.10 Translating the specification description

The entire description shall be compiled into an ASN.1 module, with the <moduleName> as the name of the ASN.1 module and AUTOMATIC TAGS.

Annex A
(informative)

Sample Encoding in MSTF

A.1 MSTF Sample

The following shows an MSTF specification intended to demonstrate all features of MSTF. A DNF specification generated by a compiler from the MSTF follows.

TITLE	In VehiculeNavigationSystemCommunicationDeviceMessageSetModule								
IMPORTS	BeaconImports								
EXPORTS	BeaconExports								
	// Defines a Normal Value								
*	AnyLabel	item							
	// Defines Months of the Year								
*	MonthsofTheYear	||							
*	*	january							
*	*	february							
*	*	march							
*	*	april							
*	*	may							
*	*	june							
*	*	july							
*	*	august							
*	*	september							
*	*	october							
*	*	november							
*	*	december							
	// Defines All Types								
*	AllTypes								
*	*	integer	I						
*	*	float	F						
*	*	bit	Bit						
*	*	alpha	A						
*	*	octet	O						
*	*	bool	Bool						
*	*	open	Hole FIRST						
*	*	open2	Hole SECOND						
*	*	enum1	DaysOfTheWeek						
*	*	enum2	MonthsOfTheYear						
*	*	enum3	College						
*	*	enum4	Seasons						
	// Defines Message Set								
*	In VehiculeNavigationSystemCommunicationDeviceMessageSet								
*	*	m1L	**IF (m1L)**		**// Dynamic Link Times Traffic Message Center (TMC)**				
*	*	*	messageCode	I 16	**// General use by TMC or RS beacon**				
*	*	*	sequenceNumber	I 6					
*	*	*	timeStamp	I 17					
*	*	*	dataFormatCode	I 3	**// Congestion levels, link travel times,**				
*	*	*			**// travel times deltas, or**				
*	*	*			**// times/speed factors (ratio).**				
*	*	*	numberSegsPerLink	I 0...6	**// 5-minutes segments. 1st - Currents**				
*	*	*			**// 2nd = Predicted 5 minutes later, etc.**				
*	*	*	locationRefArea	O 8	**// Standard location reference**				
*	*	*	linksData	{}	0 8	**// Standard location reference for link**			
*	*	*			I 2				
*	*	*			causeCode	I 6	[]		**// = not specified**
*	*	*			{I}	**// 5 minute intervals**			
*	*	*				**// Repeated numberSegsPerLinkTimes**			
*	*								
*	*	m2L	**ELSEIF(m2L)**		**// Dynamic Link Times from Roadside (RS) beacons**				
*	*	*	messageCode	I 16	**// for use by RS beacons**				
*	*	*	sequenceNumber	I 16					
*	*	*	timeStamp	I 17	**// 1 second resolution**				
*	*	*	beaconPairID	BeaconPairType					
*	*	*	dataFormatCode	I 3	**// Congestion levels, link travel times**				
*	*	*			**// travel times deltas, or time/speed**				
*	*	*			**// factors (ratios)**				
*	*	*	numberSegsPerLink	I 0...6	**// 5-minute segments**				
*	*	*	linksData	{}					
*	*	*		locationRefLink	O 8				
*	*	*		causeCodeType	I 2	**// 0 = not specified**			
*	*	*			causeCode	I 6	[]		
*	*	*		travelInfo5	{I}	**// 5 minute intervals**			
*	*	*				**// Repeated number SegsPerLink Times**			
*									
*	*	m3L	**ELSEIF(m3L)**			**// Request for Temporary Vehicle ID**			
*	*	*	messageCode	I 16		**// from Vehicle to TMC or RS beacon**			
*	*	*	transactionNo	I 16					
*									
*	*	m4L	**ELSEIF(m4L)**		**// General Text Messages**				
*	*	*	messageCode	I 16					
*	*	*	messagePriority	I 4					
*	*	*	typeInfoCode	I 16					
*	*	*	message	A 1...120	**// can be ASCII or oriental characters**				

A.2 DNF after the change (Conversion from A1)

The following shows the result of conversion from MSTF (A1).

```
----DNF Descriptions----
In-VehicleNavigationSystemCommunicationDeviceMessageSetModule
Imports BeaconImports
Exports BeaconExports
-- Defines a Normal Value
AnyLabel ::= item
-- Defines Months of the Year MonthsOfTheYear = january | february | march | april | may | june | july |
august | september | october | november | december;
-- Defines All Types
AllTypes ::= (integer I
              float F
              bit Bit
              alpha A
              octet O
              bool Bool
              open Hole FIRST
              open2 Hole SECOND
              enum1 DaysOfTheWeek
              enum2 MonthsOfTheYear
              enum3 College
              enum4 Seasons
);
In-VehicleNavigationSystemCommunicationDeviceMessageSet ::=
-- Dynamic Link Times from Traffic Message Center (TMC)
< m1L ( messageCode             I 16, -- General use by TMC or RS beacon
    sequenceNumber              I 6,
    timeStamp                   I 17,
    dataFormatCode I 3,    -- Congestion levels, link travel times,
                           -- travel time deltas, or
                           -- time/speed factors (ratios).
    numberSegsPerLink I 0..6, -- 5-minute segments. 1st - Current;
                              -- 2nd = Predicted 5 minutes later, etc.
    locationRefArea O 8,      -- Standard location reference
    linksData
             {
                  ( locationRefLink O 8, -- Standard location reference for link
                  causeCodeType I 2,     -- 0 = not specified
```

```
                              [ causeCode I 6 ],
                          travelInfo5               { I }    -- 5-minute intervals
                                                             -- Repeated numberSegsPerLink times
                         )
                    }
                  )
 |
 -- Dynamic Link Times from Roadside (RS) beacons
 m2L ( messageCode              I 16,          -- for use by RS beacons
       sequenceNumber           I 6,
       timeStamp                I 17,          -- 1 second resolution
       beaconPairID             BeaconPairType,
       dataFormatCode           I 3,           -- Congestion levels,link travel times,
                                               -- travel time deltas, or time/speed
                                               -- factors (ratios).
       numberSegsPerLink        I 0..6,        -- 5-minute segments
       linksData
           {
                 (locationRefLink O 8,
                 causeCodeType I 2,            -- 0 = not specified
                        [ causeCode I 6 ],
                 travelInfo5      { I }        -- 5-minute intervals
                                               -- Repeated
 numberSegsPerLink times
                   )
           }
    )
 |
 -- Request for Temporary Vehicle ID
 m3L ( messageCode I 16, -- from vehicle to TMC or RS beacon
       transactionNo                           I 16
    )
 |
 -- General Text Messages
 m4L ( messageCode      I 16,
       messagePriority  I 4,
       typeInfoCode      I 16,
       message          A 1..120 -- Can be ASCII or oriental characters
       )
 >;
```

Annex B
(informative)

ASN.1 Reserved Words

The following words are reserved words in ASN.1, and cannot be used as identifiers in the language described in this document. They appear in section 9.16 of document ISO/IEC 8824-1:1998. They are included here to make this document more nearly self-contained.

ABSENT	EMBEDDED	INTERSECTION	SEQUENCE
ABSTRACT-SYNTAX	END	IS0646String	SET
ALLENUMERATED	EXCEPT	MAX	SIZE
APPLICATION	EXPLICIT	MIN	STRING
AUTOMATIC	EXPORTS	MINUS-INFINITY	SYNTAX
BEGIN	EXTERNAL	NULL	T61String
BIT	FALSE	NumericString	TAGS
BMPString	FROM	OBJECTTRUE	TelexString
BOOLEAN	GeneralizedTime	ObjectDescriptor	TYPE-IDENTIFIER
BY	GeneralString	OCTET	UNION
CHARACTER	GraphicString	OF	UNIQUE
CHOICE	IA5String	OPTIONAL	UNIVERSAL
CLASS	IDENTIFIER	PDV	UniversalString
COMPONENT	IMPLICIT	PLUS-INFINITY	UTCTime
COMPONENTS	IMPORTS	PRESENT	VideotexString
CONSTRAINED	INCLUDES	PrintableString	VisibleString
DEFAULT	INSTANCE	PRIVATE	WITH
DEFINITIONS	INTEGER	REAL	

Annex C
(informative)

A Comparison of DNF to ASN.1

C.1 Comparison of Types

The international standard Abstract Syntax Notation One (ASN.1) is a very powerful tool for describing syntax. It is also a rather complex language, and reading and writing specifications in ASN.1 can be difficult. Its generality and comprehensiveness make it difficult to write specifications without understanding a number of concepts. There are a large number of built-in types, including over a dozen different types of strings. There are seven ways of sub-typing, and an entire table specifying which methods of sub-typing are applicable to which simple types. Types can be constrained in very general ways. The whole subject of tags can be very confusing (although it can be avoided by using the AUTOMATIC TAGS option). These choices give the specification writer a large amount of control over the exact data allowed and how it will be encoded. However, it is necessary to understand these options to write even simple specifications.

DNF (Descriptor Normal Form) was introduced to make it easier to write simple specifications. It is based on regular expressions. There are three basic ways to create complex components out of simpler components:

— make a sequence of components;

— choose one out of a number of components;

— repeat a component an arbitrary number of times.

ASN.1 also allows these three operations (called SEQUENCE, SEQUENCE OF and CHOICE types), but also provides a large number of other choices.

DNF provides a subset of the simple types available in ASN.1, and for each it allows at most two ways of sub-typing. Even for these types, it sometimes does not allow all of the options that ASN.1 provides.

A summary of the various types that ASN.1 defines and their equivalent DNF type (or notation) is given below. Features with no equivalent have "N.E." in the DNF column.

Table C.1 — Simple data types

ASN.1	DNF	Limitations
Boolean	Bool	
integer	I	NamedNumbers not allowed
enumerated	enumerated	
real	F	Only base 10, but standard decimal notation is allowed in DNF.
Bit string	Bit	NamedBitList not allowed, only 'B notation allowed for bistring constants
Octet string	O	Only 'H notation allowed for octet string constants
Null	NULL	Only allowed in choice types
tagged types	N.E.	DNF uses Automatic tags
object identifier	N.E.	
embedded-pdv	N.E.	
external	N.E.	
character string	A	All DNF strings are UTF8String types in ASN.1. ASN.1 provides a dozen other choices for restricted character strings, and allows unrestricted character
generalized time	N.E.	
universal time	N.E.	
object descriptor	N.E.	
open	N.E.	

Table C.2 — Complex data types

ASN.1	DNF	Limitations
sequence	(x, y)	COMPONENTS OF has no equivalent OPTIONAL uses notation [a] DEFAULT uses notation [a = x]
SEQUENCE-OF	{X}	
set	N.E.	
set-of	N.E.	
choice	<x l y>	Only Automatic tags allowed
selection	N.E.	
constrained	N.E.	
exception identifier	N.E.	
element set specification	N.E.	Except for Subtype Elements (Table 3)

Table C.3 — Subtype elements

ASN.1	DNF	Limitations
single value	value	Constants that appear in DNF are translated into a single-value type.
contained subtype	N.E.	
value range	low..high	I, F
size constraint	size	Bit, O, A types. (Fixed size) For I converted to a range constraint.
	min..max	Bit, O, A types. (Range of sizes)
permitted alphabet	N.E.	
type constraint	N.E.	
inner subtyping	N.E.	

C.2 User-defined types

ASN.1 also provides the ability to define types that can be used in the description of other types. In fact, types can be defined in terms of themselves (recursive definitions). Originally, DNF did not allow this. Because of this, DNF descriptions tended to be tabular descriptions, with component descriptions nested within other component descriptions. ASN.1 descriptions tend to give names to sub-components and to define them separately rather than nest them.

The current version of DNF has moved closer to ASN.1 in the sense that it now allows type definitions. DNF can now name sub-components and define them separately, as can ASN.1. These type definitions can be recursive, as they are in ASN.1.

In terms of formal language theory, the original DNF could describe regular languages. The ability of ASN.1 to define types recursively gave it the power to describe context-free languages. The current DNF also has type definitions which can be recursive, so it can also describe context-free languages.

The flavour of the current DNF is similar to Extended Backus-Nauer Form (EBNF). EBNF, as popularized by Wirth1) in his description of the language Pascal, is a standard way to describe languages.

In conclusion, it is probable that when users specify simple languages (regular languages), they will find it most convenient to nest constructs as is done in the original DNF. They will define common sub-components separately, but recursive definitions will not be needed and will probably be rare. For more complex languages (context-free languages), recursion will be needed and is available.

As the tables above show, ASN.1 has some construct that corresponds to each construct in DNF, and many others besides. In some sense, DNF gives an alternate syntax to a subset of ASN.1. In doing so, it masks a large numbers of details and reduces the number of possible choices. If the constructs in ASN.1 that have no equivalent construct in DNF are not needed, the DNF description should be easier for an inexperience writer to formulate, or for a program to generate automatically from a spreadsheet. If more powerful constructs are needed, then it is necessary to use ASN.1.

ICS 03.220.01; 35.240.60

Price based on 31 pages