National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

ASNST:

ABSTRACT SYNTAX NOTATION-ONE

SUPPORT TOOL

by

CHERYL ANN CLEGHORN

A M. Sc. Thesis

submitted to the School of Graduate Studies and Research

in partial fulfilment of the requirements for the

Master of Computer Science degree

University of Ottawa

Ottawa, Ontario

Canada

September, 1988

© Cheryl Ann Cleghorn, Ottawa, Canada, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-46739-8

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENTS

I would like to thank Dr. Hasan Ural for his dedication in supervising this thesis. I would also like to thank Zbigniew Koperzcak for his invaluable assistance with system details, and Souheil Gallouzi who has been helpful with the Prolog implementation.

I am indepted to the Canadian Commonwealth Scholarship and Fellowship Plan and to the Government of Trinidad and Tobago without whose financial support this program would not have been possible.

ABSTRACT

The Abstract Syntax Notation One (ASN.1) enables Application Layer Protocol Data Units (PDUs) to be defined in terms of a collection of data types. Using these built-in and defined data types and their associated encoding rules, Application Layer PDUs can be encoded for transmission across the underlying communication network. Received PDUs can also be checked to see if they conform to their logical structures specified in ASN.1 and can subsequently be decoded and represented in a user-oriented format.

An ASN.1 support system (ASNST) is described. ASNST is based on inverted execution of encoding rules associated with the ASN.1 notation and the Yacc and Lex tool provided under Unix. The system utilizes a Prolog implementation of the encoding rules which provides the capability of both generating and recognizing instances of ASN.1 built-in and defined data types. A User interface allows the construction of PDUs and PDU definitions thus removing the syntax requirements imposed on the user by the ASN.1 notation and the Yacc and Lex grammar.

Possible uses of this system are:

a) construction and validation of a PDU definition which specifies the logical structure of a particular PDU in ASN.1

b) construction and manipuation of PDUs conforming to their PDU definitions

c) encoding and decoding PDUs based on their definitions in ASN.1

d) automatic generation of sample PDUs using default values for parameter fields

e) automatic validation of PDUs with respect to their definitions.

TABLE OF CONTENTS

1. INTRODUCTION 1
  1.1 Background and Related Work 1
  1.2 Motivation and Contributions of the Thesis 2
  1.3 Outline of the Thesis 6

2. ASN.1 AND ASSOCIATED ENCODING RULES 7
  2.1 Abstract Syntax Notation-One 7
  2.2 An Overview of ASN.1 Encoding Rules 15

3. ASNST: ABSTRACT SYNTAX NOTATION-ONE SUPPORT TOOL 22
  3.1 Objectives and Capabilities of ASNST 22
    3.1.1 DEFINITION TREE 23
    3.1.2 VALUE TREE 25
    3.1.3 DEFINITION-TO-PARSETREE 27
    3.1.4 GENERATE-RECOGNIZE 28
      3.1.4.1 GENERATE: CONSTRUCTION OF R FROM A GIVEN C 30
      3.1.4.2 RECOGNIZE: CONSTRUCTION OF C FROM A GIVEN R 32
    3.1.5 PARSETREE-TO-SAMPLE 33
    3.1.6 DEFINITION-TO-SAMPLE 34
    3.1.7 EF-TO-IF 34
    3.1.8 IF-TO-EF 35
    3.1.9 The Editor 36
  3.2 User Interface of ASNST 37
  3.3 Implementation Details of ASNST 40
    3.3.1 The Editor 42
      Edit_pdu_def 42
        Createdef 43
        Editdef 44
      Edit_pdu_sample 45
        Createsample 46
        Editsample 47
    3.3.2 Requests 48
      DEFINITION-TO-PARSETREE 48
      PARSETREE-TO-SAMPLE 49
      GENERATE-RECOGNIZE 49
      DEFINITION-TO-SAMPLE 51
      EF-TO-IF 51
      IF-TO-EF 52

4. EXAMPLES 53

5. USES OF ASNST 61
  5.1 Interactive Designer Interface 61
  5.2 Encoder-Decoder 62
  5.3 Interactive Test Constructor 63
  5.4 Interactive Tester Interface 63
  5.5 Automatic Tester 64

6. CONCLUSION 65

REFERENCES 69

Appendix A USER'S GUIDE TO ASNST 70
A.1 INTRODUCTION 70
A.2 GETTING STARTED 70
A.3 FUNCTIONS PROVIDED BY ASNST 70
A.3.1 USING THE EDITOR FACILITY 71
A.3.1.1 EDITING PDU DEFINITIONS 72
Creating a PDU Definition 73
Editing a PDU Definition 74
A.3.1.2 EDITING PDU OCCURRENCES 76
Creating a PDU Occurrence in EF 77
Editing a PDU Occurrence in EF 79
A.3.2 USING THE REQUESTS FACILITY 80
A.3.3 Locating a File Specified by the User 84
A.3.4 Storing Data Generated by ASNST 85
A.4 EXITING THE SYSTEM 86

LIST OF FIGURES

Figure 3.1.1: Basic Modules of ASNST 23
Figure 3.2.1: Features of the Main Menu Provided by ASNST 38
Figure 3.2.2: Functions Provided in Editor Menu 38
Figure 3.2.3: Features Provided by Edit PDU Definition Menu 38
Figure 3.2.4: Features Provided by Edit PDU Occurrence Menu 38
Figure 3.2.5: Functions Provided in Requests Menu 39
Figure 3.3.1: Edit_pdu_def Window 43
Figure 3.3.2: Edit_pdu_sample Window 46

LIST OF TABLES

Table 2.1.1: Definition of ASN.1 Data Types 10
Table 2.2.1: Identifier Class Bits (Bits 8-7) 16
Table 2.2.2: Identifier Form Bit (Bit 6) 17
Table 2.2.3: Summary of ASN.1 Built-in Data Types 20
Table 2.2.4: Summary of ASN.1 Defined Data Types 21

1. INTRODUCTION

1.1 Background and Related Work

Programs in different computer systems need to communicate with each other in order to perform specific tasks. The set of rules that govern the orderly exchange of data among communicating programs is called a communication protocol. The International Standardization Organization (ISO) has provided a layered protocol heirarchy called Open Systems Interconnection (OSI) - Basic reference model [1], which provides a framework for communication protocols.

In this model, the highest layer is the Application Layer. The Application Layer Protocol governs the orderly exchange of data between two application layer entities. Communication between two Application Layer protocol entities is facilitated by the exchange of Application Layer protocol data units (PDUs) over the service provided by the Presentation Layer. Each PDU consists of various fields which may in turn consist of subfields. Each field (subfield) may be represented by a data type that stands for a class of data items. Usually, the rules concerning allowed values for fields of PDUs are relatively complex and constitute the major portion of Application Protocols.

Abstract Syntax Notation One (ASN.1) defines a standard notation for the presentation syntax of Application Layer PDUs [2,3]. It provides a number of built-in and defined data types which can be used to specify the logical structure of PDUs. New

data types can be created from these ASN.1 built-in and defined data types. Each ASN.1 data type has an associated encoding rule [4] which determines the representation of a data item of the corresponding data type for transmission through the underlying communication service. Encoding rules for ASN.1 data types are specified in [4].

Recently some ASN.1 support tools have been proposed [5,6,7]. Each of these tools is basically an encoder/decoder with some editing facilities. Since these editing facilities are operating on some internal representation of PDU occurrences, all of these tools require the users familiarity with the encoding rules associated with the ASN.1 notation.

1.2 Motivation and Contributions of the Thesis

The ASN.1 notation allows the definition of Application Layer PDUs in terms of the data types for the fields of the PDUs. These may be standard ASN.1 data types or other structured data types conforming to the ASN.1 standards. Structured data types are defined in terms of other, possibly structured, data types. The definitions of the structured data types must be supplied as part of the PDU definition. The definition of a PDU is complete when all the referenced data types are expressed in terms of standard ASN.1 data types. It is, therefore, not always possible to conceptualize the logical structure of a PDU by reading its ASN.1 definition. In order to simplify the conceptual view of a PDU definition, there is a need to replicate the PDU definition

given in ASN.1 into some other data structure that traces the relationship among the various fields of the PDU.

A tree representation preserves the order of the fields in the defined PDU and at the same time provides a means of directly tracing the relationships among the subfields. Hence the need for a system which converts a PDU definition given in ASN.1 to the equivalent tree representation (denoted definition tree).

The specification of values for PDU occurrences follows a format similar to that of the PDU definition and can therefore pose a similar problem in terms of conceptualizing the logical structure. It is therefore reasonable to represent the PDU occurrences, given in external form[1], in the form of a tree (denoted value tree) similar to the tree representing the PDU definition. However, because of the way in which PDUs are defined in ASN.1 (as structured data types consisting of simpler (primitive) data types) there will not be a value in the value tree for each node corresponding to the counterpart node of the definition tree.

The motivation, therefore, is to find a simpler structure with which we can represent the values specified in the PDU occurrences given in external form. The solution is a linear tree which is basically a list of values with each node in the list having a pointer to the associated primitive data type of the definition tree. These primitive data type nodes will be the

[1]The external form (EF) of a PDU occurrence refers to a PDU whose values are specified according to the ASN.1 notation

terminal nodes of the definition tree.

It is important to note that the rules governing the allowed values for fields of PDUs are relatively complex. The set of allowed values for each field of a PDU constitute a data type for that field. Because all the structured data types can be expressed in terms of standard ASN.1 data types, and there is a clearly defined range of values for these ASN.1 data types, a system which recognizes ASN.1 data types and values for these data types is also needed. Having this system, we can reduce all the fields of the PDU definitions to a set of ASN.1 data types with the aid of the tree representation, and thus be able to recognize PDU definitions as conforming to the ASN.1 standards.

The encoded form (denoted internal form[2]) of PDU occurrences is needed for transmission over the underlying communication medium. Thus, there is the need for an encoder[3]/decoder[4] facility which will permit the transformation between the external and internal forms of PDU occurrences. This facility is provided by a direct implementation of the encoding rules associated with the ASN.1 notation [4], and included as part of the subsystem for recognizing PDUs.

With these two subsystems, it will then be possible to recognize values for PDU definitions that are submitted by users

---

[2]The internal form (IF) of a PDU occurrence refers to a PDU whose values are specified according to the encoding rules associated with the ASN.1 notation

[3]Encoding means conversion of a PDU occurrence from EF to IF

[4]Decoding means conversion of a PDU occurrence from IF to EF

or received from other sources. An added asset is to be able to generate sample PDU occurrences which can be presented to users as examples of the intent of a PDU definition and the equivalent representation as determined by the encoding rules associated with the ASN.1 notation.

Also, in order to allow users, especially new users, of the ASN.1 notation to get a general idea of the notation without the added burden of syntax requirements an Editor facility is required. This Editor should allow casual users with just a knowledge of the data types and their sequencing order for a PDU definition, to familiarize themselves with the ASN.1 notation. The Editor takes these data types and constructs the equivalent PDU definition according to the ASN.1 notation. The minimal user requirement is a knowledge of the data types and their order within the PDU definition.

In order to construct PDU occurrences in external form, the Editor will prompt the user for the values associated with the primitive data types of the PDU definition. Users are also able to alter PDU occurrences and their definition via the Editor.

The thesis aims at providing a system, ASN.1 Support Tool (ASNST) that supports Application Layer protocol design, testing and validation. ASNST provides the following functionalities:

a) construction of a PDU definition which specifies the logical structure of a particular PDU in ASN.1

b) construction and manipulation of PDU occurrences

conforming to a particular PDU definition

c) validation of PDU definitions with respect to the ASN.1 notation

d) automatic generation of sample PDU occurrences using default values for parameter fields supplied during the definition of the PDU in ASN.1

e) automatic validation of PDU occurrences with respect to their definitions

f) conversion of sample PDU occurrences from a user-oriented format (External Form) to the form associated with the ASN.1 encoding rules (Internal Form)

g) conversion of sample PDU occurrences from Internal Form to External Form for presentation to the user.

Functionalities a) and b) are provided by an Editor accompanying ASNST. The other functionalities are provided by other subsystems in ASNST.

1.3 Outline of the Thesis

Chapter 2 presents a discussion on Abstract Syntax Notation One (ASN.1) along with an overview of the ASN.1 encoding rules and their implementation as Prolog clauses. In chapter 3, ASNST and its accompanying User Interface are discussed, in terms of their objectives and capabilities. Chapter 4 gives examples of the features of ASNST. Chapter 5 discusses the uses of ASNST and Chapter 6 concludes the thesis. Appendix contains a user's manual for ASNST.

2. ASN.1 AND ASSOCIATED ENCODING RULES

2.1 Abstract Syntax Notation-One

Abstract Syntax Notation One (ASN.1) is a notation which enables protocol data units (PDUs), in particular those of the Application Layer, to be defined in terms of a collection of data types. ASN.1 also enables values of these data types to be specified without determining the representation, as a sequence of octets, of instances of these data types. Thus, ASN.1 provides a notation for abstract syntax definition of Application Layer PDUs.

An ASN.1 data type is either simple or structured. A simple data type is defined by specifying all the values belonging to that data type. A structured data type is defined by referencing one or more simple or other structured data type.

Identification of data types is important in ASN.1. This is because one must be able to correctly interpret the representation of a data value and in order to do so, the data type of the value represented must be known. ASN.1 achieves this by allowing the definition of a structured data type to be given in terms of simpler data types together with a specification of all the possible values of these simpler data types and forming relationships among them. The relationships among simple data types and other already defined (possibly structured) data types used to define a new structured data type can be any of the

following:

- given an ordered list of existing data types, a single value can be formed from an ordered sequence of values, one from each data type in the list. The collection of all values formed in this way gives a new data type.

- given a list of distinct existing data types, a single value can be formed as a set of values, one from each existing data type of the list. The collection of all such new values gives a new data type.

- given a single existing data type, a new value can be formed with a sequence or set of values of the given data type. All of these new values then constitute a new data type.

- given a list of existing data types, a new value is formed by chosing a value for any one of the data types in the list. The set of all new values gives a new data type.

- given an existing data type, a new data type is formed as a subset by using some structure or ordered relationship among the values of the existing data type.

In the first two types of relationships, some or all of the data types of the list may be specified as being optional or

default. In the case of an optionally specified data type a value of that data type may be omitted when forming a value for the new data type. Where a data type is specified as default, the default value of the data type must be given in the definition of the new data type. If, during the specification of a value for the new data type, a value is omitted for the defaulted data type then the default value given in the definition of the new data type is assumed.

A list of ASN.1 simple data types and structured data types formed by using the above relationships is given in Table 2.1.1. The values of the simple data types are specified along with a summary specification of values of the structured data types.

Table 2.1.1: Definition of ASN.1 Data Types

DATA TYPE	VALUE SPECIFICATION
<u>Simple</u>	
Boolean	Two distinct values: true and false
Integer	The positive and negative whole numbers, including zero
Bitstring	All distinct values formed by an ordered sequence of zero, one or more bits
Octetstring	All distinct values formed by an ordered sequence of zero, one or more octets. Each octet is an ordered sequence of eight bits
Null	A single value: null
Characterstring	This comprises five data types easily distinguished by their tags. The values of these data types are strings of characters from some defined character set. These five data types are: IA5String, NumericString, PrintableString, S61String and S100String
Time	Consists of two characterstring data types but having a limited character set sufficient for specifying time of day. The two data types are: GeneralizedTime and UTCTime
<u>Structured</u>	
Sequence	This data type is formed by referencing a fixed, ordered list of data types (some of which may be optional). Each value of the sequence data type is an ordered list of values one from each data type of the list
Sequence of	Formed by referencing an existing data type; each value of the sequence of data type is an ordered list of values from the existing data type

Set	Formed by referencing a list of distinct data types (some of which may be optional); each value of the set data type is a set of values, one from each data type of the list
Set of	Formed by referencing an existing data type; each value of the set of data type is an unordered list of values from the existing data type
Tagged	Formed by assigning a new tag to an existing data type. The values of the tagged data type are the same as the values of the data type assigned the tag. The two data types are different because of their distinct tags
Choice	Formed by referencing a fixed list of distinct data types; each value of the choice data type is a value from any one of the data types in the list
Any	This data type is similar to the choice data type with the list of data types restricted to those data types defined using ASN.1

The ASN.1 data types and the set of values associated with the data types are defined by means of production rules in [3]. For example:

```
Booleantype  ::= BOOLEAN
Booleanvalue ::= TRUE |
                 FALSE

Octetstringtype  ::= OCTETSTRING
Octetstringvalue ::= bstring |
                     hstring
```

where bstring consists of an arbitrary number (possibly zero) of

zeros and ones preceded by a single ' and followed by the pair of characters: 'B

and hstring consists of an arbitrary number (possibly zero) of the characters: 0 1 2 3 4 5 6 7 8 9 A B C D E F preceded by a single ' and followed by the pair of characters: 'H.

All ASN.1 data types are assigned a tag which is defined by the International Standard for ASN.1 [3] or by users of the notation. Different data types may have the same tag, each individual type being identified by the context in which the tag is used. Conversely, distinct tags may be assigned to multiple occurences of a single data type and thus, creating distinct data types. Tags are intended to further qualify data being sent across the communication medium. For example, suppose A is a sequence of 10 integers each of which may be optional. If we want to send only 3 elements in the sequence, then we need to associate a tag with each element in order to guide the receiving end to identify which three elements were sent. There are four classes of tags in the ASN.1 notation:

- the Universal class tag is assigned to data types defined by the International Standard for the ASN.1 notation [3]
- the Application class tag is assigned to data types defined by other Standards. For a particular Standard the Application class tag is assigned to only one data type
- the Private class tags are assigned by enterprises, not by International Standards

- the Context-specific class tags are assigned within any use of the ASN.1 notation and interpreted according to the context in which they are used.

It is not essential for the users of the ASN.1 notation to have a knowledge of tags. However, they belong to the ASN.1 notation and are intended for machine use in the encoding of values for the data types.

Universal class tags are assigned such that it is possible to deduce from the tag the top level structure of structured data types (such as SET OF, CHOICE) and the exact data type of simple data types.

All data types of the same class of tags are distinguished by a numeric identification code. A tag therefore consists of a class and its non-negative decimal numeric assignment within the class.

Some data types (in particular the Integer and Bitstring data types) are defined with an extension which assigns identifiers to specific values of the data types. These extensions are not significant to the definition of the data types. They are intended for use in specifying values of the data types. In this case, a value for the data type can be specified using one of the identifiers instead of the number. For example, using production rules:

```
Integertype ::= INTEGER  |
                INTEGER{NamedNumberList}
```

NamedNumberList ::= identifier(SignedNumber)

SignedNumber ::= number | -number

Integervalue ::= SignedNumber | identifier

where "identifier" in "Integervalue" is equal to an "identifier" in the "Integertype" sequence with which the value is associated, and represents the corresponding number.

ASN.1 also permits the naming of data types in which case the notation "NamedType" is used. In general, NamedType is defined using the production:

NamedType ::= identifier Type | Type

The associated value for the NamedType may or may not be named:

NamedValue ::= identifier Value | Value

The identifier is not a part of the data type and has no effect on the definition of the data type. The data type referenced by the NamedType is the data type contained in the NamedType sequence. It is not compulsory to associate an identifier with a value for a named type.

A distinction must be made between NamedNumber and NamedValue in the specification of values. In NamedNumber, only the "identifier" is specified as a value whereas in NamedValue a value always accompanies "identifier". In the NamedNumber, "identifier" references a value while in NamedValue "identifier" references a data type.

2.2 An Overview of ASN.1 Encoding Rules

ASN.1 allows the definition of PDUs by referencing data types and specifying the range of values for these data types. The representation of these values for transmission across the underlying communication network is determined by applying Encoding Rules associated with the ASN.1 notation. The application of these encoding rules produces a transfer syntax for values of ASN.1 data types. The encoding rules are also applied for decoding received PDUs in order to identify the data values being transferred.

Each data type in ASN.1 has an associated encoding rule. An encoded value or data item is called a data element. A data element consists of a sequence of octets (bytes) and has three basic components which always appear in the following order:

1) **Identifier** which distinguishes one data type from another and governs the interpretation of the Content

2) **Length** which specifies the length of the Content in octets and therefore determines the end of the encoding. Length has two forms; the definite and indefinite forms.

3) **Content** which is the substance of the data element, containing the primary information the data element is intended to convey. The Content distinguishes one data item from another of the same type.

Some data elements have a fourth component, the **End-of-Contents** which is used with the indefinite form of the length.

The Identifier field occupies as minimum a number of octets as possible. For data types with tags whose numeric assignment ranges from zero to 30 (inclusive) the Identifier field consists of one octet. For data types with tags whose numeric assignment is greater than or equal to 31, the Identifier field consists of a leading octet (the first octet) followed by one or more subsequent octets.

Four classes of data types are distinguished by means of the Identifier: Universal, Application, Context-specific and Private. These four classes of data types are distinguished by means of the first and second bits (bits 8-7) of the first octet of the Identifier field of the data element. The bit configurations are shown in Table 2.2.1.

Table 2.2.1: Identifier Class Bits (Bits 8-7)

CLASS	BIT ASSIGNMENT
Universal	00
Application	01
Context-specific	10
Private	11

There are two forms of data elements, namely Primitive and Constructor which correspond to simple and structured data types, respectively. A primitive data element is one whose Content is atomic, that is, cannot be further decomposed into other data elements. The Content directly represents the value of the data type. A constructor data element is one whose Content is itself a data element or a series of data elements. The Content is the complete encoding of one or more other data values. These two forms of data elements are distinguished by means of the third bit (bit 6) of the first octet of the Identifier field of a data element. This is shown in Table 2.2.2.

Table 2.2.2: Identifier Form Bit (Bit 6)

FORM	BIT ASSIGNMENT
Primitive	0
Constructor	1

The meaning of the remaining five bits (bits 5-1) of the first octet of the Identifier field depends on the number of octets the Identifier field contains. If the Identifier consists of only one octet then the remaining five bits encode the numeric identification code (ID-Code) associated with the tag of the corresponding data type. If the Identifier field consists of more than one octet, bits 5-1 of the first octet are encoded as 11111_2 and the concatenation of bits 7-1 of the subsequent octets encode the ID-Code associated with the tag of the corresponding

data type. In all but the last subsequent octets, bit 8 has value one. Bit 8 of the last subsequent octet has a value zero thus marking the end of the Identifier octets.

There are two forms of the Length field. The definite form is used in primitive encodings of data values and may also be used in constructor encodings. In this form the Length field consists of one (short form) or more (long form) octets which represent the number of octets in the Content. If the number of octets in the Content is less than or equal to 127 the short definite form, which directly encodes in one octet the number of octets in the Content, may be used. In the short form, bit 8 is necessarily zero. If the number of octets in the Content is greater than 127 the long definite form, consisting of more than one octet, may be used. In the first octet, bit 8 has the value one and bits 7-1 encode the number of subsequent octets in the length field. The concatenation of all the bits of the subsequent octets in the length field encodes the number of octets in the Content field of the data element.

The indefinite form of the Length field consists of one octet. This octet is encoded with all eight bits having value zero and indicates that the Content octets are terminated with the End-of-Content octets. The End-of-Content field, where applicable, consists of two zero octets.

The Content field consists of zero, one, or more octets and encodes the values of data types. The Content octets depend on the data type for which the data item is being represented. A

summary of the Content and first Identifier octet of the ASN.1 built-in and defined data types is given in Table 2.2.3. Table 2.2.4 gives the ASN.1 defined data types which are constructed from the built-in data types and other already defined data types.

Table 2.2.3: Summary of ASN.1 Built-in Data Types

DATATYPE	FIRST IDENTIFIER OCTET (Class	Form	ID-Code)	CONTENT
Boolean	U	P	1	one octet with all bits zero for "false" and any bit combination for "true"
Integer	U	P	2	two's complement binary number
Bit String	U	P/C	3	binary or hexadecimal digits enclosed in apostrophe followed by the capital letters "B" or "H", respectively
Octet String	U	P/C	4	textual data taking either of two forms: same as BitString or characters enclosed in quotation marks
Null	U	P	5	contents unused
Sequence	U	C	16	elements variable in number but of one type or fixed in number and possibly of different types
Set	U	C	17	members variable in number but of one type or fixed in number and possible of different types

Tagged : The Class and ID-code are specified in the tag. If the tag is implicit, the Form is the same as the data type tagged and the Content is the Content of the data type tagged; otherwise the form is Constructor and the Content is the encoding of the tagged data type

Choice : The type is chosen from a set of one or more distinct alternatives. The representation is that of the chosen alternative

Any : The type is unrestricted, that is, chosen from the set of all built-in and defined data types. The complete encoding is identical to that of an element of the assigned type

Note: The length of the data elements, except Boolean, is determined by the Content.
C means Constructor; P means Primitive;
U means Universal

Table 2.2.4: Summary of ASN.1 Defined Data Types

DATA TYPE	DEFINITION
IA5String	[UNIVERSAL 22] IMPLICIT OCTET STRING
NumericString	[UNIVERSAL 18] IMPLICIT IA5String
PrintableString	[UNIVERSAL 19] IMPLICIT IA5String
S61String	[UNIVERSAL 20] IMPLICIT OCTET STRING
S100String	[UNIVERSAL 21] IMPLICIT OCTET STRING
GeneralizedTime	[UNIVERSAL 24] IMPLICIT IA5String
UTCTime	[UNIVERSAL 23] IMPLICIT GeneralizedTime

3. ASNST: ABSTRACT SYNTAX NOTATION-ONE SUPPORT TOOL

3.1 Objectives and Capabilities of ASNST

ASNST is a system which supports various activities in Application Layer protocol development and testing. The system is centered around the manipulation of PDUs defined according to the ASN.1 notation. ASNST provides

a) encoding and decoding of PDU occurrences specified in ASN.1,

b) generation and recognition of PDU occurrences in ASN.1,

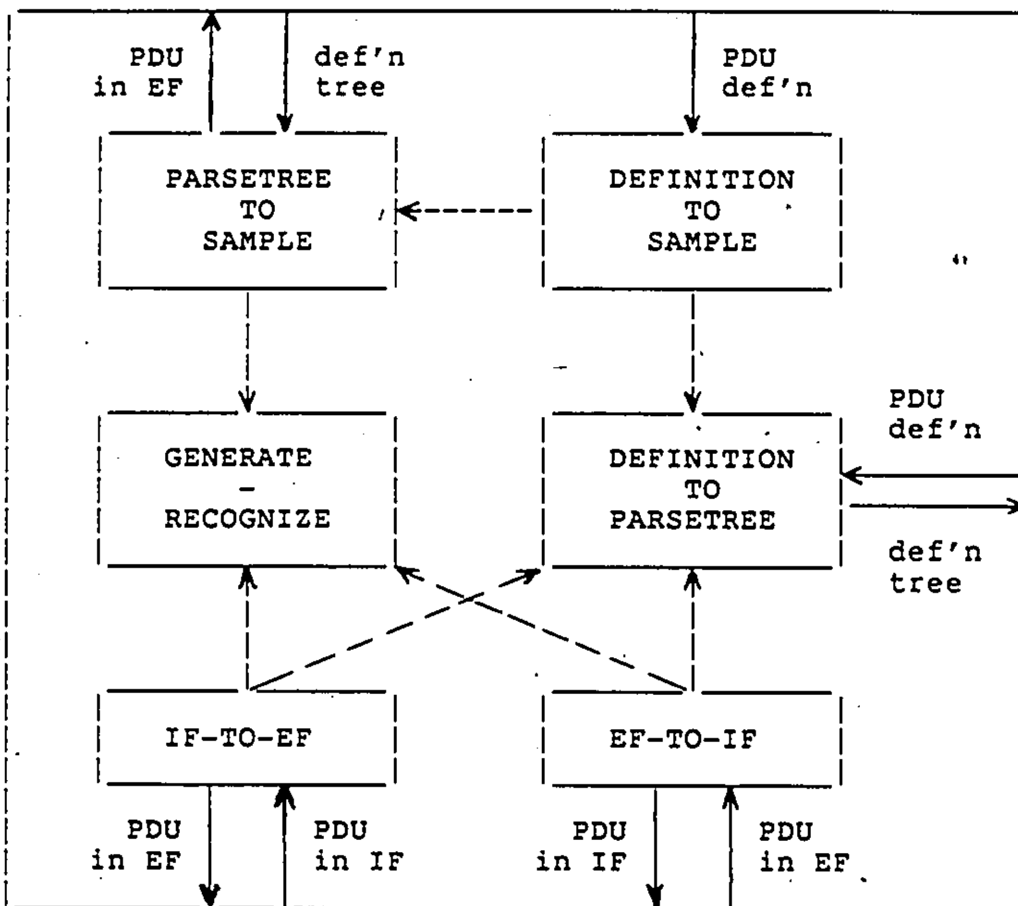
c) displaying PDU occurrences,

d) automatic validation of PDU occurrrences with respect to their definitions

An Editor facility accompanies ASNST which allows construction and manipulation of PDU definitions and PDU occurrences in a user-oriented format.

The functions of ASNST are based on PDU definitions supplied by the user. Figure 3.1.1 illustrates features a) to d) of ASNST.

The following sections describe the objectives and capabilities of ASNST in terms of the modules shown in Figure 3.1.1. Objectives and capabilities of the Editor accompanying ASNST are also described.



Solid connections: user interaction
Broken directed connections A--->B:
module A may at some point in
time invoke module B

Figure 3.1.1: Basic Modules of ASNST

3.1.1 DEFINITION TREE

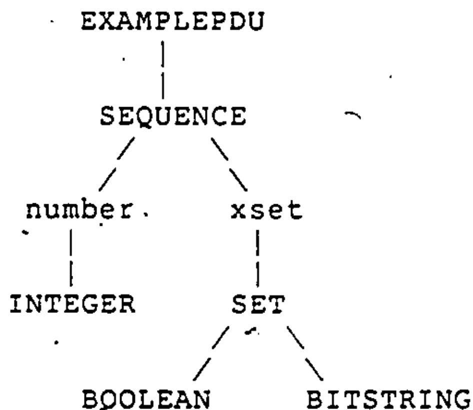
A PDU definition describes the name and type of each of its fields. Some of these fields may be complex, in which case these fields must also be described in terms of simpler fields and supplied as part of the PDU definition. These simpler fields may have to be further simplified until they can all be traced to a composition of the ASN.1 built-in and defined data types. Having

all these subdefinitions for complex fields of the PDU makes it difficult to get a clear understanding of the logical structure of the PDU. We can however represent the PDU definition in the form of a tree in order to simplify the conceptual view of the logical structure of the PDU. This tree representation is called a definition tree.

The definition tree for a PDU definition is constructed such that there is a node in the tree for each data type occurring in the PDU definition. Each datatype that is not an ASN.1 built-in or defined data type is the root of a subtree which defines that data type. Accordingly, each component of a structured data type is in a subtree under the node representing the structured data type. For example, the PDU definition

```
EXAMPLEPDU DEFINITION ::=
     BEGIN
       %EXAMPLEPDU ::= SEQUENCE{number, xset}
       %number     ::= INTEGER
       %xset       ::= SET{BOOLEAN, BITSTRING}
     END
```

will be represented by the definition tree whose nodes have the types as follows:



Here we can see that the structured or constructor data types and the non-standard ASN.1 data types form the interior nodes of the definition tree, and the simple or primitive data types form the terminal nodes of the definition tree.

3.1.2 VALUE TREE

The specification of values for PDU occurrences reflects the structure of their PDU definitions. The value of a structured data type is specified as a set of values for its component data types. The value of a non-standard ASN.1 data type is specified in terms of the value for the data type defining the non-standard data type. Consequently, a PDU occurrence will consist of subsets of values for the primitive data types defining the fields of the PDU.

In order to provide easy manipulation of PDU occurrences, a PDU occurrence is converted to a list structure called a **value tree**. The value tree is constructed in association with the definition tree for the PDU. Each node of the definition tree whose data type is primitive, should have an associated value in the value tree. Exceptions are in the case where the data type is default or optional or the data type is a component of the SEQUENCE OF or SET OF constructor data type, which can have zero repetition of values for its component data type. Each node of the value tree points to the associated node of the definition tree.

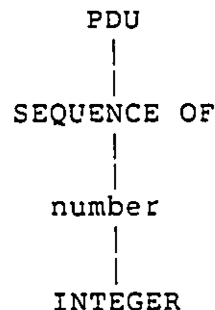
There are two types of nodes in the value tree; nodes having

actual values for primitive data types and dummy nodes. The SEQUENCE OF and SET OF data types permit any number of repetitions of values for the data types comprising the sequence or set. In order to identify a group of values as forming one repetition of the sequence or set, the group of values is preceded by a dummy node in the value tree. This dummy node points to the 'SEQUENCE OF' or 'SET OF' node in the related definition tree.

As an example, consider the following PDU definition:

```
PDU DEFINITION ::=
    BEGIN
      %PDU ::= SEQUENCE OF number
      %number ::= INTEGER
    END
```

The definition tree of this PDU reflecting all the data types in the PDU definition is:



The value tree for a PDU occurrence {5,15,34}, will consist of six nodes; three dummy nodes and three nodes having the values 5, 15, 34 respectively. The dummy nodes will point to the 'SEQUENCE OF' node and the other three nodes will have a pointer to the 'INTEGER' node in the definition tree. Representing a node of the value tree as 'datatype|value', where datatype is the data type of the associated node of the definition tree and value

is the value (NULL for dummy nodes) stored at the value tree node, the value tree will be:

SEQUENCE OF|NULL----> INTEGER|5----> SEQUENCE OF|NULL----> INTEGER|15 ----> SEQUENCE OF|NULL ----> INTEGER|34

3.1.3 DEFINITION-TO-PARSETREE

DEFINITION-TO-PARSETREE takes an ASN.1 definition of a PDU and constructs the corresponding definition tree for the PDU. Before the construction takes place, the PDU definition is checked whether it conforms to the ASN.1 notation. Only when this check gives positive result is the definition tree constructed.

All the other modules of ASNST utilize the definition trees of PDUs. If a definition tree of a PDU is not available for a specific module, that module calls DEFINITION-TO-PARSETREE to make the definition tree for the PDU available.

A definition tree is made accessible (as opposed to transparent) to users of ASNST thus enabling them to construct before hand all the definition trees for PDU definitions that will be used in the current session of ASNST. A list of all definition trees constructed in the current session of ASNST is kept, in order to avoid duplication of trees. Before a tree is built by DEFINITION-TO-PARSETREE, the list is checked to see if the tree already exists for the PDU definition under consideration. If it does exist, then a new one is not constructed; otherwise the construction of the tree proceeds and the list is updated. All definition trees remain available to

the system for the duration of the current session of ASNST.

3.1.4 GENERATE-RECOGNIZE

GENERATE-RECOGNIZE is an implementation of the encoding rules for the built-in and defined data types associated with the ASN.1 notation. These encoding rules are implemented in terms of Prolog clauses. A Prolog clause defining the encoding rule of a datatype X can be used for two main purposes:

1) for generating the representation of a data element Y of type X,
2) for checking whether a given representation of a data element Y is of type X.

These Prolog clauses are of the form datatypeX([L,C],R) for simple data types and datatypeX(T,[L,C],R) for structured data types. DatatypeX, the name of the clause, is the ASN.1 datatype, L is the length (in decimal) of the value in the content field of the representation R, C is the content (in alphanumeric) of R, and (in the case of structured data types) T is a list of datatypes that form components of datatypeX.

A Prolog datatypeX clause is invoked by specifying the name of the clause (i,e., datatypeX), and the list T if datatypeX refers to a structured datatype, together with instantiations of some of the parameters L, C and R. The instantiated parameters are recognized by GENERATE-RECOGNIZE and if successful, values are generated for the uninstantiated parameters. Some examples of possible uses of these clauses are illustrated below using the

clause "integer([L,C],R)" defining the encoding rule for integer data type.

The invocation "integer([L,10],R)" will recognize 10 as an integer and generate [[0,2],[0,1],[0,A]] as the value of R. The value of R is the ASN.1 representation of the data element 10 which is of integer data type. Here, the parameter L is instantiated to "1" which is the length (in decimal) of the value of the content field of the representation R.

The invocation "integer([L,C],[[0,2],[0,1],[0,E]])" will check whether the given representation is for a data element of integer data type. Having successfully recognized R, the following is generated: L = 1; C = 14.

The invocation "integer([1,C],R)" may randomly generate some integer value of length 1, e.g., C = 6 and R = [[0,2],[0,1],[0,6]].

An example of the invocation of a structured data type, using the clause "sequence(T,[L,C],R)" defining the encoding rule for sequence data type, is the invocation "sequence(integer,[L,C],R)" which produces: L = 7; C = [[1,34],[2,345]]; and

R = [[3,0],[0,7],[[[0,2],[0,1],[2,2]],[[0,2],[0,2],[0,1,5,9]]]].

If the values supplied for the parameters with the invocation are not valid with respect to the encoding rule, any attempt to match those values with the representation of a data element will fail. Clauses of the form datatypeX([L,C],R) or datatypeX(T,[L,C],R), where all the parameters are

uninstantiated, are termed **empty clauses** and are used to generate sample occurrences of data values and/or data elements of ASN.1 data types.

3.1.4.1 GENERATE: CONSTRUCTION OF R FROM A GIVEN C

Whenever the value of R, corresponding to a given C for a data element in a PDU occurrence is needed, both the value tree for the PDU occurrence and the corresponding definition tree are used. The definition tree is necessary since we need to know how the values in the value tree relate to the data types of the PDU definition. First, C is retrieved from the value tree by simultaneously traversing the definition and value trees. Since C may be complex for a structured data element, consisting of a series of [L,C] values, we need to know how the [L,C] grouping occurs. Also, for each subtree of the definition tree whose root is an ASN.1 standard data type, there will be an associated [L,C].

For example, in order to retrieve C for a data element in the given PDU occurrence in section 3.1.2, we will use the given definition tree and the value tree. After initiating the definition of C, the definition tree will be traversed, until the node pointed to by the current (in our example the first) node of the value tree is reached. At the first node of the definition tree, we instantiate C to '['. Since the datatype is SEQUENCE OF we need to check if there are any values for the

sequence. The current node of the value tree points to the 'SEQUENCE OF' node of the definition tree, so there is a value and C is compound. Note that if the value tree node did not point to the 'SEQUENCE OF' node of the definition tree, it would have indicated zero repetition which is valid, and C would have been null, i.e., C = []. The second node of the definition tree is not a standard ASN.1 data type, so we move to the third node which is a standard ASN.1 data type. So we expect a value from the value tree to be associated with this node, unless the datatype is OPTIONAL or DEFAULT. At this point we will have C = '[[_,5]' substituting the value at the value node. Recall that [_,5] is the [L,C] of the first component of the data element whose type is SEQUENCE OF.

The values for L in these [L,C] groupings are not of much use. This is because they only give the lengths of subfields of the data element and not the length of the contents of the entire data element. These L's are therefore replaced by the Prolog anonymous variable, '_', the underscore. The values for L's can, however, easily be calculated using the values of R's once they are obtained.

The next value node points to the 'SEQUENCE OF' node of the definition tree so there is another repetition of the sequence. This repetition produces C = '[[_,5],[_,15]'. The other two nodes of the value tree produces C = '[[_,5],[_,15],[_,34]'. We have now exhausted all the nodes in the value tree under the "SEQUENCE OF" node, so the definition of C is completed. The

final result is C = '[[_,5],[_,15],[_,34]]'.

After C is retrieved from the value tree, the value of R corresponding to C is obtained by submitting the clause "sequence(integer,[L,[[_,5],[_,15],[_,34]]],R)" to GENERATE-RECOGNIZE.

3.1.4.2 RECOGNIZE: CONSTRUCTION OF C FROM A GIVEN R

In order to obtain C, corresponding to a given R for a data element, R must be instantiated in the clause datatypeX([L,C],R) or datatypeX(T,[L,C],R). A PDU occurrence in IF contains the value of R in the form needed by the clause, thus this value is simply transferred to the clause. For example, for the following PDU in IF

IFPDU{ [[3,0],[0,3],[[[0,2],[0,1],[2,3]]]] }END

whose corresponding PDU definition is that given in section 3.1.2, the value for R will be

R = [[3,0],[0,3],[[[0,2],[0,1],[2,3]]]].

After R is retrieved from the PDU in IF, the value of C corresponding to R is obtained by submitting the clause "sequence(integer,[L,C],[[3,0],[0,3],[[[0,2],[0,1],[2,3]]]])" to GENERATE-RECOGNIZE.

3.1.5 PARSETREE-TO-SAMPLE

PARSETREE-TO-SAMPLE generates sample PDU occurrences for a PDU definition represented by a definition tree. This is done by generating an empty clause for the structured data type representing the PDU. This empty clause is then passed to GENERATE-RECOGNIZE for the generation of a sample PDU occurrence.

An empty clause is constructed as follows:
The data types of the data elements in the PDU occurrence to be generated are obtained by traversing the definition tree and identifying the ASN.1 data types associated with the nodes of this tree. For example, consider the PDU definition and its definition tree given in section 3.1.2. Traversing the definition tree to determine the name (datatypeX) of the clause and the list T, (if necessary) will result in the following steps:
At the first node, datatypeX is "sequence" since SEQUENCE OF is an ASN.1 datatype. SEQUENCE OF is a structured datatype, so T is to be generated from the subtree pointed to by the 'SEQUENCE OF' node. At the second node, number is not an ASN.1 data type. Thus it should point to a subtree which defines number and T is not set. At the third node, T becomes "integer" since INTEGER is an ASN.1 data type. At this point there are no more nodes to be traversed so the clause is completed as "sequence(integer,[L,C],R)".

This empty clause is then passed to GENERATE-RECOGNIZE for generation of a sample PDU occurrence. PARSETREE-TO-SAMPLE uses

the values returned by GENERATE-RECOGNIZE to produce PDU occurrences in internal form (IF) and/or external form (EF). The value of R constitutes the IF for the PDU occurrence. C consists of values for the subfields of the PDU and their lengths. PARSETREE-TO-SAMPLE constructs the EF for the PDU occurrence using only the subfield values contained in C.

To illustrate this, on sumbitting the above clause to GENERATE-RECOGNIZE we get: L = 7; C = [[1,34],[2,345]]; and R = [[3,0],[0,7],[[0,2],[0,1],[2,2]],[[0,2],[0,2],[0,1,5,9]]]. The IF will be R and the EF will be {34,345}.

3.1.6 DEFINITION-TO-SAMPLE

DEFINITION-TO-SAMPLE takes a PDU definition and generates sample PDU occurrences. This module concatenates the processes of DEFINITION-TO-PARSETREE and PARSETREE-TO-SAMPLE in order to accomplish its function. Thus, the user does not explicitly request that a definition tree be constructed. However, the definition tree which is constructed by this module, can be subsequently accessed by the user, since it is inserted in the list of definition trees constructed by the current session of ASNST.

3.1.7 EF-TO-IF

Given sample PDU occurrences in EF it is possible to obtain their IF representations using the EF-TO-IF module. The IF is required for exchanging PDU occurrences across the underlying

communication medium. Insisting that users of the communication medium must use this IF format restricts the types of users of the Application Layer to those who are experienced in the ASN.1 encoding notation. To avoid this, the user is allowed to represent PDU occurrences in EF.

The objective of EF-TO-IF is to generate the IF representation of a PDU occurrence, given its EF representation. This module generates the clause datatypeX(T,[L,C],R) for the given PDU occurrence. DatatypeX and T are obtained using the same definition tree traversal process as in 3.1.5. C is constructed from the value tree representing the EF form of the PDU, as described in section 3.1.4.1. EF-TO-IF then submits the clause to GENERATE-RECOGNIZE in order to generate R, which constitutes the IF of the PDU.

3.1.8 IF-TO-EF

IF-TO-EF accomplishes the reverse objective of EF-TO-IF. It is capable of accepting the IF representation of PDU occurrences and converting them to EF for presentation to users of ASNST. This is necessary so that users will be able to understand messsages received over the communication medium.

IF-TO-EF generates a clause datatypeX(T,[L,C],R) with R instantiated to the IF of the PDU. This clause is then submitted to GENERATE-RECOGNIZE for generation of C. IF-TO-EF then constructs the EF using the values of the subfields of C.

3.1.9 The Editor

The Editor allows users of ASNST to enter and subsequently edit PDU definitions and PDU occurrences in EF into the system without users having to concern themselves with syntax requirements. PDU definitions and PDU occurrences in EF are handled by two subsystems in the Editor; edit_pdu_def and edit_pdu_sample.

In creating a new PDU definition, the user enters the name and type of the fields and subfields for the PDU definition, along with other information such as optional, default and tagged fields. The Editor then uses this information to construct the PDU definition in the format needed for later usage by the system. The Editor recognizes fields whose types are not ASN.1 built-in or defined data types and automatically prompts the user for a definition of these types.

PDU definitions can also be altered using the Editor. Fields can be deleted or changed. Component fields of the same structured data type can be exchanged with each other and types can be changed. New components of existing fields or new fields can also be inserted in the PDU definition.

PDU occurrences in EF can be entered into the system via the Editor. In this case, the corresponding PDU definition is displayed and the user is prompted for values that represent the lowest level of the field definitions. The data types for these lowest level fields will inevitably be simple ASN.1 data types. Users can edit PDU occurrences in EF by replacing values for the

fields of the PDU.

3.2 User Interface of ASNST

All the modules described in section 3.1 (with the exception of GENERATE-RECOGNIZE which cannot be explicitly called by users) are invoked via a user interface associated with ASNST. This user interface provides menus to the user and the menus inform the user of the functions that can be utilized at any given time.

The first menu, shown in Figure 3.2.1, is the main menu and introduces ASNST as comprising an Editor and a Request based subsystem. At this point the user will select, via the numerical assignment, which of the two facilities he wishes.

1. INTERACTIVE EDITOR
2. REQUESTS
3. QUIT

Figure 3.2.1: Features of the Main Menu Provided by ASNST

The user may quit the system by selecting option 3.

Selection of option 1 of Main Menu transfers control to the Editor. With the Editor the user may edit PDU definitions or PDU samples in EF. The specific task is accomplished by selecting one of the options of the Editor Menu shown in Figure 3.2.2.

1. PDU DEFINITION

2. SAMPLE PDU IN EXTERNAL FORM

Figure 3.2.2: Functions Provided in Editor Menu

For each of these options the user can then create a new PDU definition or PDU occurrence, respectively, or edit already existing PDU definitions or PDU occurrences in EF. Again menus are used as guides to the user. These menus are shown in Figure 3.2.3 and Figure 3.2.4.

(C)REATE A NEW PDU DEFINITION

(E)DIT AN EXISTING PDU DEFINITION

Figure 3.2.3: Features Provided by
Edit PDU Definition Menu

(C)REATE A NEW PDU OCCURRENCE IN EXTERNAL FORM

(E)DIT AN EXISTING PDU OCCURRENCE IN EF

Figure 3.2.4: Features Provided by
Edit PDU Occurrence Menu

The desired feature is selected by entering the corresponding letter enclosed in parenthesis.

Selection of option 2 of Main Menu transfers control to the subsystem containing the modules described in 3.1.1 to 3.1.8 and brings out the menu shown in Figure 3.2.5 for the user to select any of the functions provided by these modules.

1. CONSTRUCT INTERNAL TREE FROM PDU DEFINITION
2. GENERATE SAMPLE PDUs IN INTERNAL FORM AND/OR EXTERNAL FORM, GIVEN INTERNAL TREE
3. GENERATE SAMPLE PDUs IN INTERNAL FORM AND/OR EXTERNAL FORM, GIVEN PDU DEFINITION (This is choices 1 and 2 combined)
4. GENERATE PDUs IN INTERNAL FORM GIVEN PDUs IN EXTERNAL FORM
5. GENERATE PDUs IN EXTERNAL FORM GIVEN PDUs IN INTERNAL FORM

Figure 3.2.5: Functions Provided in Requests Menu

Selection of one of the numerical assignments associated with each function activates the desired module. Each of these modules will further prompt the user for information relating to the desired function. This information includes names of PDU definitions or PDU occurrences and names of files containing them.

3.3 Implementation Details of ASNST

ASNST is implemented in the C programming language and runs under the Unix operating system on the VAX 780 . A skeleton pseudo-code of ASNST is given as follows:

```
Main Program:
     select = get_main_selection
     while (select != quit)
          {
          switch (select)
               case 'editor'  : rc = Editor;
               case 'request' : rc = Requests;
          end;
          if (rc == quit)
               select = quit
          else
               select = get_main_selection
          }; /* end while */


Editor :
     select = get_editor_selection
         while (select != quit)
          {
          switch (select)
               case 'definition' : edit_pdu_def;
               case 'sample' : edit_pdu_sample;
```

```
            case  'exit'  : return(continue);
        end;
        select = get_editor_selection
        }; /* end while */
    return(quit);


Requests:
    select = get_request_selection
    while (select != quit)
        {
        switch (select)
            case 'def_tree' : DEFINITION-TO-PARSETREE;
            case 'tree_spl' : PARSETREE-TO-SAMPLE;
            case 'def_spl'  : DEFINITION-TO-SAMPLE;
            case 'ext_int'  : EF-TO-IF;
            case 'int_ext'  : IF-TO-EF;
            case 'exit'     : return(continue);
        end;
        select = get_request_selection
        };  /*  end while  */
    return(quit);
```

The main program consists of a while loop which transfers control to the Editor or the Requests routine, until quit is specified. The Requests routine is responsible for providing access to the modules given in Figure 3.1.1. The main program

constructs and displays the Main Menu, using get_request_selection, and then waits for the user's selection. On processing the selection, the Main Menu is redisplayed and the process repeats. The loop exits (or is never entered) when the user chooses to quit. The user can opt to quit via the menu or at some time during the execution of one of the called routines.

3.3.1 The Editor

The Editor is also implemented in a loop which transfers control to routines for editing PDU definitions or PDU occurrences in EF. An Editor Menu allows the user to make the desired selection. From the menu, the user can also choose to quit the system or to exit the Editor subsystem. Exit results in the resumption of the main program. The Editor allows editing PDU definitions by calling the routine edit_pdu_def, and PDU occurrences are edited by calling edit_pdu_sample.

Edit_pdu_def:

The skeleton structure of edit_pdu_def is

```
select = get_edit_selection
switch(select)
     case 'create' : createdef;
     case   'edit' : editdef;
     case 'return' : return;
end;
```

The edit_pdu_def routine consists of a menu handler and a case statement. The menu handler, get_edit_selection, allows the user to select whether a new PDU definition is to be created or an already existing PDU definition is to be edited. The user may also choose to return to the Editor. According to the user's selection, the case statement transfers control to the related subroutine. If the user indicates that he wants to create a new PDU definition, control is transfered to the createdef routine. If an existing PDU definition is to be edited, control is transferred to the editdef routine.

Createdef:

Createdef construcвts a window through which the user enters the name of the PDU and the data types which define the PDU. The form of the window is given in Figure 3.3.1.

PDU_NAME: ____________

TYPE_NAME: ________ TYPE: ________ TAG: ID: IMPLICIT?

COMPONENTS	NAMED TYPE?	TYPE	TAGGED	OPTIONAL?	DEFAULT?
________	___	______	___	___	___
________	___	______	___	___	___

Figure 3.3.1: Edit_pdu_def Window

Createdef prompts the user for the information in the window by highlighting the desired field. If the type is a named type,

TYPE_NAME is the name of the type currently being defined; otherwise TYPE_NAME and TYPE are the same and represent the data type. If the type currently being defined is constructor then createdef prompts the user for information related to the component(s) of the constructor data type. All data types that are not ASN.1 built-in or defined data types must be defined. Createdef keeps a list of all such data types. When the current window is completed, new windows are displayed for the definition of these data types, in the order in which they appear in the list. TYPE_NAME of the new window is then assigned the type to be defined. Before a new data type is added to the list, the list is scanned to see if the data type has previously been added to the list. If it already exists then it is not entered a second time.

Whenever a window is completed, createdef constructs the equivalent type definition required by the ASN.1 notation. The first window will allow the construction of the header for the definition. When the definition is completed the END of the definition is added as required by the ASN.1 notation.

Editdef:

Editdef is designed to allow editing of already existing PDU definitions. It uses the window of Figure 3.3.1 to display the information given by a PDU definition. The PDU definition is converted to its equivalent definition tree and editdef traverses the definition tree in order to retrieve the information for

displaying in the window. A copy of the PDU definition given in ASN.1 is kept in the data structure temp_def. Temp_def is used to make the changes requested by the user. On completion of these changes, if the user commits his actions, the original definition is replaced by that of temp_def.

Editdef prints at the bottom of the window, a list of editing options permitted by the Editor. The options are (D)elete, (R)ename, e(X)change, (I)nsert, (A)ppend and (S)kip. The user selects the desired option for editing the highlighted field, by typing in the letter given in parentheses above. Editdef then prompts the user for new information according to his choice; in the case of the (S)kip option, the Editor highlights the next field of the definition window for editing. The changes are made to the definition tree as specified by the user.

In order to make the equivalent change to the PDU definition given in ASN.1 the system editor, ed, of the Unix operating system is used. Editdef writes corresponding ed commands to a file cmdfile. When the user completes his changes on the definition, ed is run on temp_def using the commands in cmdfile.

Edit_pdu_sample:

This routine performs similar functions as edit_pdu_def. The skeleton structure is

```
select = get_edit_selection
switch (select)
```

```
        case 'create' : createsample;
        case 'edit' : editsample;
        case 'return' : return;
    end;
```

The user's option, select, is obtained by the menu handler accompanying edit_pdu_sample. If select equals create, control is transferred to the createsample routine. If select equals edit, control goes to editsample. On selecting return edit_pdu_sample returns to Editor.

Createsample:

Createsample, like createdef, uses a window to obtain information from the user. The form of the window is given in Figure 3.3.2.

```
                 PDU_NAME: xxxxx

   TYPENAME: xxxxx        TYPE: xxxxx      VALUE: __________

     COMPONENTS           TYPE          VALUE

        xxxxx            xxxxx          ____________

                         xxxxx          ____________
```

Figure 3.3.2: Edit_pdu_sample Window

Createsample traverses the definition tree of the associated PDU definition, in order to obtain the data represented by xxxxx. According to the data type, createsample prompts the user for the

corresponding value. Only primitive data types require values to be specified. As values are obtained, the PDU occurrence is constructed according to the ASN.1 notation.

Editsample:

Editsample allows editing of existing PDU occurrences. The user issues along with the name of the PDU occurrence, its corresponding PDU definition name. The definition tree for the PDU is then retrieved by editsample. Next, the value tree is constructed for the PDU occurrence and both trees used to display the data contained in the PDU occurrence. A value is expected for all primitive data types (except, possibly in the case of optional data types). So the original PDU occurrence would have a value reference in the value tree for all the primitive data types. Therefore, all that can be done in editing a PDU occurrence are changes to the already existing values. Insertions, deletions, appends and exchanges are thus meaningless.

The window of Figure 3.3.2 is also used by editsample. The data xxxxx are obtained from the definition tree and the associated values are obtained from the value tree. As values are changed, the values in the value tree are also changed to reflect the user's action.

A copy of the PDU occurrence in ASN.1 is kept in the data structure temp_ext_spl. The editor ed is used to make the equivalent changes to temp_ext_spl, in a similar manner to

editdef. Editing commands for ed are written to cmdfile.

3.3.2 Requests

Requests consists of a menu handler for obtaining the user's options and a loop transferring control to the routine that will handle the particular option. When control returns to Requests, another user option is obtained and the loop continues. The loop exits when the user chooses to quit or to return to the main menu.

DEFINITION-TO-PARSETREE:

The structure of DEFINITION-TO-PARSETREE is

```
found = check_tlist;
if (found)
    return
else
    buildtree;
```

Whenever a definition tree is constructed, the name of the definition is entered in a list, tlist, which keeps track of all the definition trees constructed in the current ASNST session. When the user requests that a definition tree be constructed, DEFINITION-TO-PARSETREE checks tlist to see if the definition tree already exists, by calling check_tlist. If it exists, check_tlist returns true to the variable 'found', else false is returned. If 'found' is true, then DEFINITION-TO-PARSETREE is completed. Otherwise the definition tree is constructed using

the Unix system routines Yacc and Lex [8]. Lex takes the PDU definition given in ASN.1 and breaks it up into tokens to be passed to Yacc. The tokens are the type names, data types, values and keywords. Yacc then uses the tokens to instantiate the fields of the nodes of the definition tree as it is built.

PARSETREE-TO-SAMPLE:

PARSETREE-TO-SAMPLE uses the definition tree to construct empty Prolog clauses of the form datatypeX(T,[L,C],R) for GENERATE-RECOGNIZE. DatatypeX is generated by a traversal of the definition tree. At each node of the definition tree, PARSETREE-TO-SAMPLE performs the following function

```
if (node_type == ASN.1_built-in_or_defined)
     update_datatypeX
else
     next_node
```

When datatypeX is completed, PARSETREE-TO-SAMPLE completes the empty clause datatypeX([L,C],R) and then submits it to GENERATE-RECOGNIZE for generation of the sample.

GENERATE-RECOGNIZE:

GENERATE-RECOGNIZE is implemented in Prolog and provides the capability of generating and recognizing instances of data elements of all types in ASN.1. In this module, the encoding rules associated with the ASN.1 data types are implemented in terms of clauses of the form:

```
datatypeX([L,C],R):-
(   var(R),(            /*  R IS NOT GIVEN  */
            (var(C), /*  C IS NOT GIVEN  */
              generate_C(L,C),
              assign_check_value_X(L,C,R))
          /*             GENERATE R FROM C  */
            ;
            (nonvar(C),  /*  C IS GIVEN  */
              assign_check_value_X(L,C,R))
          /*  CHECK C, GENERATE R FROM C  */
           )
)
;
(nonvar(R),(                /*  R IS GIVEN  */
            (var(C), /*  C IS NOT GIVEN  */
              assign_check_value_X(L,C,R))
          /*  CHECK R, GENERATE C FROM R  */
            ;
            (nonvar(C),  /*  C IS GIVEN  */
              assign_check_value_X(L,C,R))
          /*             CHECK R, CHECK C  */
            )
 ).
```

where L, C and R stand for length (in decimal), content (in alphanumeric) and representation of the data element,

respectively. Generate_C generates a value for C and assigns its length to L. Assign_check_value_X functions in the following way: if the value of L is given then it is used to check whether R and C are of this length or to generate C of this length. If the value of L is not given, then it is determined.

DEFINITION-TO-SAMPLE:

This routine utilizes DEFINITION-TO-PARSETREE and PARSETREE-TO-SAMPLE. DEFINITION-TO-PARSETREE is first called to make the definition tree for the PDU definition available. When the definition tree is obtained, a call is made to PARSETREE-TO-SAMPLE to construct the empty clause required to generate the sample. GENERATE-RECOGNIZE is then called to complete the process.

EF-TO-IF:

The structure of EF-TO-IF is

```
root = get_definition_tree;
if (root != NULL)
     {
     construct_value_tree;
     generate_datatypeX;
     add_content;
     GENERATE-RECOGNIZE;
     }
else
```

error

EF-TO-IF consists of a call to get_definition_tree which makes the definition tree for the PDU available. If the PDU definition exists then get_definition_tree returns the root of the tree. Otherwise, NULL is returned and an error message printed.

The value tree for the external representation of the PDU, is then constructed. This is done by traversing the definition tree and identifying the values of the fields of the PDU occurrence with the terminal nodes of the definition tree. Generate_datatypeX constructs datatypeX for GENERATE-RECOGNIZE as explained in the section on PARSETREE-TO-SAMPLE. The content C of the clause is then added and the clause completed. GENERATE-RECOGNIZE is then called to generate the internal representation of the PDU.

IF-TO-EF:

IF-TO-EF is implemented in a similar manner as EF-TO-IF. However, instead of add_content, add_R is called. This adds the internal representation of the PDU to the clause to be submitted to GENERATE-RECOGNIZE. IF-TO-EF then scans C and extracts the values specified for the subfields of the PDU. These values are then used to construct the required EF.

4. EXAMPLES

This chapter gives examples of the various functionalities of ASNST. The functions of the Requests subsystem will be illustrated using examples constructed by the Editor. The functions of the Editor will be illustrated using a simple PDU definition and a PDU occurrence for that definition.

The use of the Editor cannot be illustrated as a series of the exact user/system interactions. This is because extensive cursor movements are involved and cannot be properly captured on manuscript. Even the use of the Unix 'script' feature and the printing of the scripted file is unintelligible. Therefore the final results of the Editor facility will be printed, along with a description of what is intended.

To illustrate the Requests facility, we will use a portion of the definition of the Logical Descriptor of the SFD (Simple Formattable Document) PDU given in CCITT's Red Book [9]. The portion of the Logical Descriptor being used is:

```
LogicalDescriptor ::= SEQUENCE{LogicalObjectType,
                               LogicalDescriptorBody}
LogicalObjectType ::= INTEGER{document (0), paragraph (1)}
LogicalDescriptorBody ::= SET{pageHeading [3] IMPLICIT
                              T61String OPTIONAL,
                         presentationDirectives [5] IMPLICIT
                         PresentationDirectives OPTIONAL}
PresentationDirectives ::= SET{alignment [0] IMPLICIT
```

```
                                  Alignment OPTIONAL,
                                  graphicRendition [1] IMPLICIT
                                  GraphicRendition OPTIONAL}
Alignment           ::= INTEGER{leftAligned (0),centered (2),
                                 justified (3)}
GraphicRendition ::= SEQUENCE OF GraphicRenditionAspect
GraphicRenditionAspect ::= INTEGER
```

For the Editor, some of the terms have been shortened because of the field widths defined in the program for the data types and the names. This restriction only serves to allow for the window size of 80 columns.

The PDU definition constructed by the Editor is:

```
LogDes DEFINITIONS ::=
   BEGIN
     % LogDes ::= SEQUENCE{LogObjType, LogDesBody}
     % LogObjType ::= INTEGER{ document(0), paragraph(1) }
     % LogDesBody ::= SET{pageHeading [3] IMPLICIT
                          S61String OPTIONAL,
                          presDir [5] IMPLICIT
                          PresDir OPTIONAL}
     % PresDir ::= SET{alignment [0] IMPLICIT
                       Alignment OPTIONAL,
                       graphRend [1] IMPLICIT
                       GraphRend OPTIONAL}
```

```
% Alignment ::= INTEGER{ leftAligned(0), centered(2),
                          justified(3)}
% GraphRend ::= SEQUENCE OF GraphAspect
% GraphAspect ::= INTEGER
END
```

The percentage signs preceding each subdefinition is required by Yacc and Lex in order to identify the beginning of a new subdefinition.

We will now construct a PDU occurrence for the above PDU definition, to be used for conversion to internal form. This PDU occurrence will be stored in the default file, ext_file, and given the name TestPDU. The following values will be assigned to the fields of the PDU:

FIELD	VALUE
LogObjType	document
pageHeading	page1
alignment	justified
graphRend	4,5

The PDU occurrence constructed by the Editor is:

```
TestPDU({document,{"page1",{justified,{4,5}}}})END
```

The conversion of a PDU occurrence from EF to IF will now be illustrated using TestPDU (above) constructed by the Editor.
The PDU occurrence in IF produced by EF-TO-IF is:

```
int_logdes{
[[3,0],[1,9],
       [[[0,2],[0,1],[0,0]],
        [[3,1],[1,4],
               [[[8,3],[0,5],[7,0,6,1,6,7,6,5,3,1]],
                [[8,5],[0,'B'],
                       [[[8,0],[0,1],[0,3]],
                        [[8,1],[0,6],
                               [[[0,2],[0,1],[0,4]],
                                [[0,2],[0,1],[0,5]]]]]]]]]]
}END
```

The conversion from IF to EF will be illustrated using the following PDU occurrence in IF:

```
int_logdes2{
[[3,0],[1,'C'],
       [[[0,2],[0,1],[0,1]],
        [[3,1],[1,7],
               [[[8,3],[0,5],[7,0,6,1,6,7,6,5,3,6]],
                [[8,5],[0,'E'],
                       [[[8,0],[0,1],[0,3]],
                        [[8,1],[0,9],
                               [[[0,2],[0,1],[0,4]],
                                [[0,2],[0,1],[0,5]],
                                [[0,2],[0,1],[0,'B']]]]]]]]]]
}END
```

The corresponding PDU occurrence in EF produced by IF-TO-EF

is:

ext_logdes2({1,{"page6",{3,{4,5,11}}}})END

We will now illustrate the generation of sample PDU occurrences using the following simplified version of LogDes, called SimLogDes (this version was also constructed using the Editor):

```
SimLogDes DEFINITIONS ::=
   BEGIN
     % SimLogDes ::= SEQUENCE{LogObjType,
                                LogDesBody}
     % LogObjType ::= INTEGER
     % LogDesBody ::= SET{pageHeading S61String,
                              presDir PresDir}
     % PresDir ::= SET{alignment Alignment,
                           graphRend GraphRend}
     % Alignment ::= INTEGER
     % GraphRend ::= SEQUENCE OF GraphAspect
     % GraphAspect ::= INTEGER
   END
```

The PDU occurrence generated in EF by DEFINITION-TO-SAMPLE is:

EF_Sample({-218942006,{"SILth",{26107,{29268,162}}}})END

The corresponding PDU occurrence generated in IF by DEFINITION-TO-SAMPLE is:

```
IF_Sample{
[[3,0],[2,3],
      [[[0,2],[0,9],[D,F,B,E,C,2,B,E,3,9,9,2,F,3,3,5,C,A]],
       [[3,1],[1,6],
             [[[1,4],[0,5],[5,3,4,9,4,C,7,4,6,8]],
              [[3,1],[0,D],
                   [[[0,2],[0,2],[6,5,F,B]],
                    [[3,0],[0,7],
                         [[[0,2],[0,2],[7,2,5,4]],
                          [[0,2],[0,1],[A,2]]]]]]]]]]
}END
```

We will now illustrate the following editing features on SimLogDes:

- change the name of the PDU to 'newdef'
- change the definition of LogObjType to 'newint' and define newint as INTEGER
- exchange the two components of LogDesBody and change component 'pageHeading' to be an unnamed data type. (The Editor will then have to prompt for the definition of pageHeading; we will define it to be S61String).

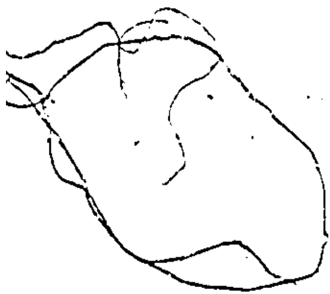
These changes result in the following PDU definition produced by the Editor:

```
newdef DEFINITIONS ::=
  BEGIN
    % newdef ::= SEQUENCE{LogObjType,
```

```
                          LogDesBody}
    % LogObjType ::= newint
    % newint ::= INTEGER
    % LogDesBody ::= SET{presDir PresDir,
                         pageHeading}
    % pageHeading ::= S61String
    % PresDir ::= SET{alignment Alignment,
                      graphRend GraphRend}
    % Alignment ::= INTEGER
    % GraphRend ::= SEQUENCE OF GraphAspect
    % GraphAspect ::= INTEGER
  END
```

We will now delete the component graphRend from PresDir. This produces:

```
newdef DEFINITIONS ::=
  BEGIN
    % newdef ::= SEQUENCE{LogObjType,
                          LogDesBody}
    % LogObjType ::= newint
    % newint ::= INTEGER
    % LogDesBody ::= SET{presDir PresDir,
                         pageHeading}
    % pageHeading ::= S61String
    % PresDir ::= SET{alignment Alignment
                     }
    % Alignment ::= INTEGER
```



END

Note that these two sets of changes could have been done in one edit session.

5. USES OF ASNST

ASNST is an invaluable tool which can be used in the area of Application Layer protocols design, testing and validation. This chapter outlines some of the uses of ASNST in this context.

5.1 Interactive Designer Interface

As an interactive designer interface, ASNST aids in the specification and validation of application layer protocols. PDU definitions and rules governing the allowed values of PDU fields usually constitute a major portion of the specification of an application layer protocol. The inclusion of correct definitions of PDUs is facilitated by the system as follows:

1) The protocol designer may use the Editor to construct PDU definitions in ASN.1 by selecting "PDU DEFINITION" of the Editor Menu. The Editor prompts the designer for the names and data types of the fields and subfields of the PDU definition. Data types that are not ASN.1 built-in or defined data types are noted by the Editor. The Editor subsequently prompts the designer for the definitions of these data types.

   Since, during the specification of a PDU definition, any violation of the ASN.1 notation is communicated to the designer, the use of ASNST, specifically the Editor, reduces the risk of specifying PDu definitions not conforming to the ASN.1 notation.

2) Any PDU definition conforming to the ASN.1 notation may then be passed to the DEFINITION-TO-PARSETREE module for construction of a definition tree corresponding to this definition.

3) The designer may then choose to activate DEFINITION-TO-SAMPLE module in order to generate sample PDU occurrences in external form and/or internal form satisfying the existing PDU definitions. The examination of PDU occurrences, generated by ASNST, constitutes a valuable informal validation of part of the protocol specification.

4) Whenever needed, modifications to the PDU definitions and PDU occurrences in EF may also be done using the Editor.

5.2 Encoder-Decoder

As an Encoder-Decoder, ASNST aids in the encoding and decoding of PDU occurrences according to the PDU definitions given in ASN.1 and included in the corresponding protocol specification. Encoding of a PDU involves transforming the PDU from its external form (stream-oriented form suitable for transfer of information between application programs) to the internal form (data-element oriented form suitable for transfer of information through the communication medium). Decoding of a PDU involves the transformation of the PDU from its internal form to its external form. Note that in either case, the PDU occurrence is verified whether its structure and the values of its fields are consistent with the corresponding PDU definition.

5.3 Interactive Test Constructor

As an interactive test constructor, ASNST facilitates the construction of particular PDUs to be used in testing and debugging an application protocol implementation under test (IUT). A test designer may use the Editor to construct, in advance, PDUs that will be sent to the IUT by selecting "SAMPLE PDU IN EXTERNAL FORM" of the Editor menu. The Editor prompts the test designer for values of PDU fields of a chosen PDU definition and employs these values to build the PDU in EF. Constructed PDUs are stored externally for further use. The Editor also helps in the modification of existing PDUs which were externally stored. A particular variation of this usage is the construction of service primitives that the user of an application layer protocol entity sends to or receives from the entity.

5.4 Interactive Tester Interface

As an interactive tester interface, ASNST supports testing and debugging of an application layer protocol implementation under test. PDUs constructed (or selected from previously constructed ones) by the tester, are sent to the IUT by invoking the EF-TO-IF module of ASNST. This module is invoked when the tester chooses "GENERATE PDUs IN INTERNAL FORM GIVEN PDUs IN EXTERNAL FORM" of the Requests menu. PDUs received from the IUT can be displayed on the screen in a readable format using the IF-TO-EF module. This module checks the output of the IUT (that is,

the received PDU in IF) for conformance to the corresponding PDU definition and converts it to EF for further analysis by the tester. This module of ASNST is invoked by selecting "GENERATE PDUs IN EXTERNAL FORM GIVEN PDUs IN INTERNAL FORM" of the Requests menu.

5.5 Automatic Tester

As part of an automatic tester, ASNST may aid in construction of test sequences in terms of PDUs and service primitives and automated application of these sequences to an IUT.

6. CONCLUSION

ASN.1 (Abstract Syntax Notation One) provides a set of built-in and defined data types for the definition of the logical structure of application layer protocol data units (PDUs). The notation supports a set of encoding rules associated with the data types defined in the ASN.1 standard. These encoding rules provide for the representation of PDUs in a form suitable for transmission through the underlying communication medium.

Implemented as Prolog clauses, the encoding rules provide the capability of both generating and recognizing instances of the ASN.1 built-in and defined data types. These clauses have been used as the basis of a support system, ASNST, for generating and recognizing PDUs based on their definition in ASN.1, and for encoding and decoding PDUs being transmitted across the communication medium.

Functions of ASNST include:

- construction of definition trees which reflect the logical structure of PDUs,
- generation of PDU occurrences in EF and IF from their PDU definition or from their definition tree,
- conversion of PDU occurrences from EF to IF and vice versa (i.e., encoding and decoding, respectively).

ASNST also has an Editor facility which allows construction and manipulation of PDU definitions in ASN.1 and PDU occurrences in EF.

ASNST can be used in applications which require PDU generation, recognition and conversion facilities such as:

- an interactive tester in which PDUs must be constructed and sent to the implementation under test (IUT) and in which PDUs received from the IUT must be recognized and displayed in a readable or processible format,
- communication software development where PDUs are to be defined in ASN.1 and included in the protocol specification,
- encoding and decoding PDUs.

ASNST has been implemented on the Vax 780 computer in C and Prolog and runs under the Unix operating system. The only part of ASNST that is implemented in Prolog is the GENERATE-RECOGNIZE module.

ASNST does not permit the construction of the definition tree from a PDU definition with data types that are DEFAULT and accompanied by a value. This is because any element or member of the SEQUENCE or SET data types can be defined as DEFAULT, and this element or member can be a structured data type. This means that the value for an element or member of a SEQUENCE or SET can be a complex value.

The default value must be declared as a token for Lex. It is difficult to specify such a token since the value may consist arbitrarily of strings, numbers and other complex values. For example, the subdefinition

```
EXAMPLE ::= SEQUENCE{ complex DEFAULT value, INTEGER}
complex ::= SET{BITSTRING, names, INTEGER}
```

names ::= SEQUENCE OF S61String

should have 'value' replaced by

{'014F'H,{"TOM","JONES"},2}

ASNST therefore only allows the definition of the PDU without specifying a default value.

The Editor facility allows the creation of PDU definitions with tags, named numbers, OPTIONAL and DEFAULT data types. However, the editing of such PDUs once they are created is restricted. Full editing capabilities are provided with PDU definitions that do not have data types with named numbers or tags, or OPTIONAL or DEFAULT data types. This is more of a restriction than a drawback. All that is required to alleviate this restriction, is to include case statements for named number, the tag possibilities, and OPTIONAL and DEFAULT, in each of the routines for the edit features. The strings in the ed commands will have to be changed to reflect the various cases. This has been done for the tagged data type that is to be renamed, in order to show what the case statements and the corresponding ed command look like.

Each of the components of a structured data type must be on a separate line of the data file and commas separating two components must be on the same line as the first component. This restriction is because the ed commands must be specific. PDU definitions constructed by the Editor follow this restriction.

As pointed out earlier, part of ASNST (i.e., the GENERATE-RECOGNIZE module) is implemented in Prolog. The interface

between C and Prolog is facilitated by submitting clauses to Prolog for execution. This interface is currently slow and needs improvement.

Suggested improvements are a follow up of the restrictions mentioned above:

a) Each node of the definition tree should have a field for default values. So after a definition tree is constructed, code can be included to allow the user to enter the default values interactively. These values can then be stored in the definition tree at the node whose data type is marked DEFAULT. These values will, however, have to be re-entered each time the definition tree is constructed. An alternative is to use a table to store default values for each PDU definition.

b) The Editor can be improved to a full scale Editor by allowing for the tags, named numbers, OPTIONAL and DEFAULT data types as indicated above. The amount of cases involved will be large because there are four classes of tags ,(IMPLICIT or EXPLICIT).

c) New interpreters and compilers for Prolog are now emerging with facilities for interfacing with other programming languages (e.g., ALS Prolog by Applied Logic Systems). ASNST can therefore be improved by porting the Prolog program to one of these new Prolog interpreter or compiler, and inserting the necassary conversion code into the C program.

REFERENCES

[1] Zimmermann, H., "OSI Reference Model - The ISO model of architecture for Open Systems Interconnection", IEEE Trans. on Communication, Com-28 (4), pp. 425-432, 1980.

[2] CCITT Draft Recommendation X.409. Message Handling Systems: Presentation Transfer Syntax and Notation.

[3] Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), ISO 8824, 1987.

[4] Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), ISO 8825, 1987.

[5] Bochmann, G.v. et al, "Application layer testing and ASN.1 support tools", Proc. IEEE GLOBECOM '86 conference, pp 767 - 771, 1986.

[6] Bessette S; Bochmann G.v., "Implementation issues for the ISO file transfer, access and management protocol", Proceedings of the Thirteenth Biennal Symposium on Communications : 2-4 June,1986, Queen's University at Kingston, pp C.4.16 - C.4.18.

[7] Pope A.R., "Encoding CCITT X.409 Presentation Transfer Syntax", Computer communication review, pp 4 - 10, October 1984.

[8] Kernigham B.W., and Pike R, The Unix Programming Environment

[9] CCITT Red Book, Data Communication Networks Message Handling Systems

Appendix

A USER'S GUIDE TO ASNST

A.1 INTRODUCTION

Abstract Syntax Notation One Support Tool (ASNST) is a menu-driven system for manipulating protocol data units (PDUs) defined in ASN.1. ASNST is implemented in the C programming language and runs under the Unix operating system on the VAX 780. Activities include:

- construction and editing of PDUs and PDU definitions
- conversion of PDUs from External Form (EF) to Internal Form (IF) and vice versa
- generation of PDU occurrences from their PDU definitions given in ASN.1

This appendix comprises a user's guide to ASNST.

A.2 GETTING STARTED

The C progams comprising the system are compiled and linked into the object code ASNST. The system is thus invoked by issuing the command ASNST, at the Unix system prompt.

A.3 FUNCTIONS PROVIDED BY ASNST

On starting the system, the following message appears:

WELCOME TO ASNST

****

Hit any key to continue

At this point the user enters any character and the main menu appears:

M A I N   M E N U

==================

1. INTERACTIVE EDITOR
2. REQUESTS
3. QUIT

YOUR CHOICE (NUMBER) ===>

The response to the choice prompt is one of the three numbers which precede the menu entries.

Option 1: 'INTERACTIVE EDITOR' allows the user to enter the Editor facility

Option 2: 'REQUESTS' allows the user to enter the facility for PDU conversion and generation.

Option 3: 'QUIT' provides exit from the ASNST system

Other : Any other entry results in an error message and redisplaying of Main Menu.

A.3.1 USING THE EDITOR FACILITY

On entering the Editor by selection of option 1 of the main menu, the following Editor Menu is displayed:

INTERACTIVE EDITOR

==================

YOUR EDITOR ACCEPTS:

1. PDU DEFINITION
2. SAMPLE PDU IN EXTERNAL FORM

   (You can also)

3. RETURN TO MAIN MENU
4. QUIT

YOUR CHOICE (NUMBER) ===>

The response to the choice prompt is one of the four numbers which precede the menu entries.

Option 1: 'PDU DEFINITION' provides access to an editor subsystem which allows the editing of PDU definitions.

Option 2: 'SAMPLE PDU IN EXTERNAL FORM' provides access to an editor subsystem which allows the editing of PDU occurrences in External Form (EF).

Option 3: exits the Editor facility and returns the user to the main menu

Option 4: exits the ASNST system

A.3.1.1 EDITING PDU DEFINITIONS

Option 1 of the Editor menu transfers control to a subsystem for editing PDU definitions. Editing PDU definitions involves creating new PDU definitions or editing already existing definitions. The system thus displays the menu:

EDIT PDU DEFINITION MENU

========================

(C)reate a new PDU definition

(E)dit an existing PDU definition

(R)eturn

Please enter choice:

The choice is the letter enclosed in parentheses for each of the options. The choice is recognized in both uppercase and lowercase. Choosing Return (r or R) returns you to Editor menu. PDU definitions are created or edited via the PDU definition window:

PDU_NAME: ________

TYPE_NAME: ____	TYPE: ____		TAG:	ID:	IMPLICIT?
COMPONENTS	NAMED TYPE?	TYPE	TAGGED	OPTIONAL?	DEFAULT?
____	__	____	__	__	__
____	__	____	__	__	__

Creating a PDU Definition

Selecting c or C of the 'Edit PDU definition' menu results in the empty PDU definition window being displayed. The system prompts the user for the information for each field of the definition window by moving the cursor to the respective fields

and sounding a bell. For the definition of structured data types, the user signifies the end of entries for the components by hitting the [ENTER] key, when the system prompts for another component. Questions which require a yes/no answer are answered using the letters y,Y,n or N.

When the creation of the PDU definition is completed, ASNST stores the definition in the file indicated by the user, or in the default PDU definition file. (See **Storing data generated by ASNST**).

Editing a PDU Definition

On selecting e or E of the 'Edit PDU definition' menu, the system displays the message:

Enter name of PDU definition:

The user then enters the definition name.

The system respond with:

Enter name of file containing 'defname':

Where 'defname' is the name just entered by the user.

The user then enters the name of the file, and the system attempts to retrieve it. (See **Locating a file specified by the user**). When the file is successfully retrieved the system searches it for the PDU definition specified by the user. If it is not found, the following error messages are displayed:

'defname' does not exist in 'filename'

Do you want to try another file? (yes/no)

If the user responds with 'yes', the file retrieval process

repeats; otherwise the subsystem exits. If the PDU definition exists, the editing proceeds.

The PDU definition is displayed through the PDU definition window. Each of the fields is highlighted and the user responds with one of the following fixed editing option displayed at the bottom of the window:

(D)elete (R)ename e(X)change (I)nsert (A)ppend (S)kip

where

Delete : deletes a PDU definition, data type or components of data types

Rename : changes the name of a PDU definition or a data type

eXchange: interchanges two components of the same data type

Insert : adds a new component in front of the current component of a data type

Append : adds a new component after the current component of a data type

Skip : activates editing of the next field of the window

The user's response for an option is the letter enclosed in parentheses. Both uppercase and lowercase are recognized.

Note that not all of the options apply to all the fields shown in the window. When the user selects an inappropriate option he is thus informed and allowed a different option. The user is prompted for new information according to the selected option.

When the user has completed the editing process, the system prompts:

Do you wish to commit these changes to 'filename'? (Y/N)

where 'filename' is the name of the file from which the PDU definition was retrieved. The user responds with y, Y, n or N accordingly. A response of y or Y causes the original PDU definition to be deleted from the file. The response n or N leaves 'filename' unchanged. Note that in both cases, the definition tree corresponding to the PDU definition is deleted from the system. ASNST then returns to 'Edit PDU Definition' menu.

A.3.1.2 EDITING PDU OCCURRENCES

Option 2 of the Editor menu transfers control to a subsystem for editing PDU occurrences. Editing PDU occurrences involves creating new PDU occurrences or editing already existing PDU occurrences. The system thus displays the menu:

EDIT PDU OCCURRENCE MENU

========================

(C)reate a new PDU occurrence in EF

(E)dit existing PDU occurrence in EF

(R)eturn

Please enter choice:

The choice is the letter enclosed in parentheses for each of the options. The choice is recognized in both uppercase and lowercase. Choosing Return (r or R) returns you to Editor menu.

PDU occurrences are created or edited via the PDU occurrence window:

```
PDU_NAME: xxxxx

TYPENAME: xxxxx      TYPE: xxxxx      VALUE: ________

COMPONENTS           TYPE             VALUE

xxxxx                xxxxx            __________

                     xxxxx            __________
```

Creating a PDU Occurrence in EF

The user is allowed to create a new PDU occurrence by selecting option c or C of the 'Edit PDU occurrence' menu. The PDU definition is needed to determine which field of the PDU occurrence window takes actual values and which ones consist of a set of values. So the system displays the prompt:

Enter name of associated PDU definition:

The user responds with the PDU definition name.

If the system has previously located the PDU definition (since its last edit) then the creation of the new PDU occurrence proceeds. Otherwise the system prompts the user for the name of the file containing the definition. The user directs the system to the location of the file (see **Locating a file specified by the user**) and the PDU definition is retrieved.

Having determined the existence of the PDU definition, the system prompts the user as follows:

Enter name to be given to PDU occurrence in EF:

and the user enters the name denoted 'sample_name'. The system then prompts the user for the values of the various fields of the PDU occurrence. Each data type the window is highlighted and the user has two alternatives:

If the data type is constructor then the system displays the massage:

[ENTER] for submenu

The user responds by hitting the [ENTER] key.

The system then displays the fields for that constructor data type and prompts the user for field values.

If the data type is primitive, the system displays the message:

Enter value

The user enters the value at the position indicated by the cursor.

Whenever the user does not want to enter a value, he simply hits the [ENTER] key. If such an action is permitted, the prompt goes to the next field. Otherwise an error message:

value required

is printed and the system waits for a value.

When the creation of the PDU occurrence in EF is completed, ASNST stores the PDU occurrence in the file indicated by the user or the default file for PDU occurrences in EF (See **Storing data generated by ASNST**). ASNST then returns to 'Edit PDU Occurrence' menu.

Editing a PDU Occurrence in EF

The user activates this facility by selecting option e or E of the 'Edit PDU Occurrence' menu. The system then displays the prompt:

Enter name of external PDU occurrence

and the user responds with the PDU occurrence name. The system then prompts for the file containing the PDU occurrence and retrieves the PDU occurrence in EF (see **Locating a file specified by a user**).

Having located the PDU occurrence, the system prompts for and retrieves the associated PDU definition as in the previous section.

The system displays the PDU fields and values in the PDU occurrence window and prompts the user according to the data type for each field.

If the data type is constructor, the system prompts:

[ENTER] for submenu

The user hits the [ENTER] key and the fields and value for that data type are displayed in the window for editing.

If the data type is primitive, its value is highlighted and the user is allowed to respond to the question:

Change value? (Y/N)

The user responds with y or Y for yes and n or N for no. If the response is y or Y, the system clears the value field and prompts for a new value. Otherwise the system continues with the

next field.

When the user has completed the editing process, the system prompts:

Do you wish to commit these changes to 'filename'? (Y/N)

where 'filename' is the name of the file from which the PDU occurrence in EF was retrieved. The user responds with y, Y, n or N accordingly. A response of y or Y causes the original PDU occurence to be deleted from the file. The response n or N leaves 'filename' unchanged. ASNST then returns to 'Edit PDU Occurrence' menu.

A.3.2 USING THE REQUESTS FACILITY

On entering the Requests facility by selection of option 2 of the Main Menu, the following Requests menu is displayed:

R E Q U E S T S   S U P P O R T E D

====================================

1. CONSTRUCT INTERNAL TREE FROM PDU DEFINITION
2. GENERATE SAMPLE PDUs IN INTERNAL FORM AND/OR EXTERNAL FORM, GIVEN INTERNAL TREE
3. GENERATE SAMPLE PDUs IN INTERNAL FORM AND/OR EXTERNAL FORM, GIVEN PDU DEFINITION (This is choices 1 and 2 combined)
4. GENERATE PDUs IN INTERNAL FORM GIVEN PDUs IN EXTERNAL FORM
5. GENERATE PDUs IN EXTERNAL FORM GIVEN PDUs IN INTERNAL FORM
6. RETURN TO MAIN MENU

YOUR CHOICE ('7' TO QUIT) ===>

The response to the choice prompt is one of the six numbers which precede each of the menu entries, or response 7 to quit the system. The following describes the user/system interactions subsequent to the selection of an option.

Option 1: provides access to the facility for constructing definition trees.

The system prompts:

Enter definition name:

and the user enters the definition name, defname. If the definition tree has previously been constructed (since the last edit of the PDU definition), then the process is completed. Otherwise the process continues as follows:

The system prompts:

Enter name of file containing 'defname':

The user enters the file name. The file is then retrieved (see **Locating a file specified by the user**).

If the PDU definition is not found, the following messages appear:

'defname' does not exist in 'filename'

Do you want to try another file?

The user responds 'yes' or 'no'. If the response is yes, the file retrieval process repeats. Otherwise the system returns to the Requests menu.

If the PDU definition is found, the tree construction proceeds and the system returns to the

Requests menu. Failure of this process results in an error message. At any time in this option, Return (r or R) returns you to the Requests menu.

Option 2: provides access to the facility for generating sample PDU occurrences from definition trees. The system prompts:

Enter name of definition represented by the tree:

The user responds with the name of the PDU definition. If the definition tree does not exist, the system gives the error message:

Sorry, tree does not exist

and returns to the Requests subsystem. If the tree exists, the system prompts:

Sample in (1) EF, (2) IF or (3) both?

The user enters the number preceding the option, according to his choice. The sample(s) are generated and stored as indicated in **Storing data generated by ASNST.** The system then returns to Requests menu.

Option 3: provides access to the facilities of options 1 and 2. The user/system interactions are the same as for option 1. and the final prompt of option 2. Samples are stored as indicated in option 2. After the samples are stored, the system returns to Requests menu.

Option 4: provides access to the facility for converting PDU occurrences from EF to IF. The system must first retrieve the PDU occurrence given in EF. The user/system interactions are the same as those for retrieving a PDU occurrence for editing. See **Editing an existing PDU occurrence** in section A.3.1.2.

The definition tree for the PDU is also required. The user/system interactions are the same as those for constructing a definition tree from a PDU definition (See 'option 1' of this section). The conversion from EF to IF is then performed by the system, and the PDU in IF stored by ASNST (See **Storing data generated by ASNST**). The system then returns to Requests menu.

Option 5: provides access to the facility for converting PDU occurrences from IF to EF. The system must first retrieve the PDU occurrence given in IF. The user is thus prompted as follows:

Enter name of internal PDU occurrence:

and the user must enter the name given to the PDU occurrence. The system then prompts

Enter name of file containing 'int_spl':

where 'int_spl' is the name input by the user. The system retrieves the file using the user/system interactions of 'Locating a file specified by the user'. If 'int_spl' exists, the definition tree for the PDU is retrieved as in option 4 and the conversion

is performed. The PDU in EF is stored as in

Storing data generated by ASNST

and ASNST returns to Requests menu.

Option 6: returns control to the main menu and its facilities.

A.3.3 Locating a File Specified by the User

Whenever a file specified by the user is to be accessed, the system searches the current directory (i.e., the directory containing ASNST) for that file. If the file is not in the current directory, the system responds with:

Where is 'filename' located?

And the user enters the path to the file. The system then tries to retrieve the file from the directory specified by the path. If the file is not found there, the system redisplays the path and asks the user to verify it. If the path is correct and the file is still not found an error message is given and the user given the choice of specifying another file.

Note: The file must have read permission mode set in order to be located. Otherwise the file cannot be opened for processing.

If the path is incorrect then the user is prompted for the correct path and the search continues. For each path specified, the user is given one chance to verify it.

At any point where a name is required by the system, the user can enter 'r' or 'R' to return to the Editor or Requests menu.

A.3.4 Storing Data Generated by ASNST

Using the Editor facility, ASNST generates PDU definitions in ASN.1 and PDU occurrences in EF. With the Requests facility, ASNST generates PDU occurrences in EF and in IF. These PDU definitions and PDU occurrences are stored by ASNST in files specified by the user, or in default files. The default files are:

def_file for PDU definitions,

ext_file for PDU occurrences in EF, and

int_file for PDU occurrences in IF.

Using 'dataname' as the name assigned by the user for the generated PDU definition or PDU occurrence, and 'default_file' for the name of the default file, the user/system interactions are as follows:-

The system prompts:

Enter name of file to which 'dataname' is to be added:

The user then has 2 options:

Option 1: The user can respond with a name 'filename'. If 'filename' can be opened by ASNST for writing, then 'dataname' is written to 'filename'. Otherwise the messages:

'filename' cannot be opened for writing

'dataname' written to 'default_file'

Hit [ENTER] to continue

are displayed. The user then hits the [ENTER] key in order to return to the Editor or Requests menu.

Option 2: If he wishes to use the default file, the user simply hits the [ENTER] key, instead of specifying a file name. ASNST then prints the message

'dataname' written to 'default_file'

Hit [ENTER] to continue

and returns to the Editor or Requests menu.

A.4 EXITING THE SYSTEM

ASNST is exited by selecting the Quit option of Main Menu (enter 3), the Editor menu (enter 4) or the Requests menu (enter 7).