Technical Report

DTR/MTS-00054 V0.0.5 (March-99)

Methods for Testing and Specification (MTS); A harmonised integration of ASN.1, TTCN and SDL

Key words: TTCN, SDL, ASN.1, methodology

Reference

DTR/MTS-00054 (<Shortfilename>.PDF)

ETSI Secretariat

Postal address

F-06921 Sophia Antipolis CEDEX - FRANCE

Office address

650 Route des Lucioles - Sophia Antipolis
Valbonne - FRANCE
Tel.: +33 4 92 94 42 00 Fax: +33 4 96 65 47 16

X.400

c= fr; a=atlas; p=etsi; s=secretariat

Internet

secretariat@etsi.fr
http://www.etsi.fr

Contents

Foreword

This clause contains fixed text elements for the foreword.

1 Scope

This ETSI Technical Report (TR) defines the changes that need to made to ASN.1, SDL and TTCN in order to harmonise these three languages in a consistent and compatible manner.

The technical solutions to achieve this harmonisation are based on the initial analysis of TR 101 114 [8].

This document is restricted to documenting in broad terms the changes necessary to each of the three languages. The actual changes to the relevant standards (if accepted) will need to be performed by the relevant committees in ETSI, the ITU-T and ISO. While this document offers complete technical solutions for harmonisation, it is not written in the form of detailed amendments to the relevant standards.

This TR is restricted to harmonising the versions of ASN.1, SDL and TTCN as listed in the references clauses.

2 References

[1] ITU-T Recommendation X.680 (1997): "Abstract Syntax Notation One (ASN.1): Specification of basic notation"

[2] ITU-T Recommendation X.681 (1997): "Abstract Syntax Notation One (ASN.1): Information object specification"

[3] ITU-T Recommendation X.682 (1997): "Abstract Syntax Notation One (ASN.1): constraint specification"

[4] ITU-T Recommendation X.683 (1997): "Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications"

[5] ITU-T Recommendation Z.100 (1993): "Specification and Description Language (SDL)"

[6] ITU-T Recommendation Z.105 (1995): "SDL combined with ASN.1"

[7] ISO/IEC 9646-3 (1998): "Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN)"

[8] TR 101 114 (1998): Methods for Testing and Specification (MTS); Analysis of the use of ASN.1 94 with TTCN and SDL in ETSI deliverables

3 Definitions symbols and abbreviations

3.1 Definitions

For the purposes of the present document, the following definitions apply:

ASN.1:1997: ASN.1 as defined in the 1997 ITU-T Recommendations X.680 [1], X.681 [2], X.682 [3] and X.683 [4] (Note that for historical reasons the 1997 version of ASN.1 is sometimes referred to as ASN.1 98. This term is not used in this document)

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
ATS	Abstract Test Suite
BER	Basic Encoding Rules
BNF	Backus-Nauer Form
SDL	Specification and Description Language
TTCN	Tree and Tabular Combined Notation

4 Introduction

Historically, ASN.1, SDL, and TTCN have been developed as separate standards. However, it is becoming increasingly common for all three standards to be used in a close relationship. Several tool platforms already provide some form of integrated support of these languages/notations. It is now necessary, from both a user's and a toolmaker's point of view for the standardisation process to be more closely co-ordinated.

The current versions of ISO/IEC 9646-3 [7] and Z.105 [6] define dialects of ASN.1 that are not consistent with the version of ASN.1 as defined in X.680 [1]. This document defines the changes necessary to each of these three languages/notations in order to remove these dialects and to provide a harmonised integration of the relevant standards.

The harmonisation of ASN.1, SDL and TTCN proposed in this TR is effected by proposing changes to the relevant standards in order to provide

1. a clean, unambiguous interface between ASN.1 and SDL; and
2. a clean, unambiguous interface between ASN.1 and TTCN;

NOTE: This document assumes that there is no language interface necessary between SDL and TTCN.

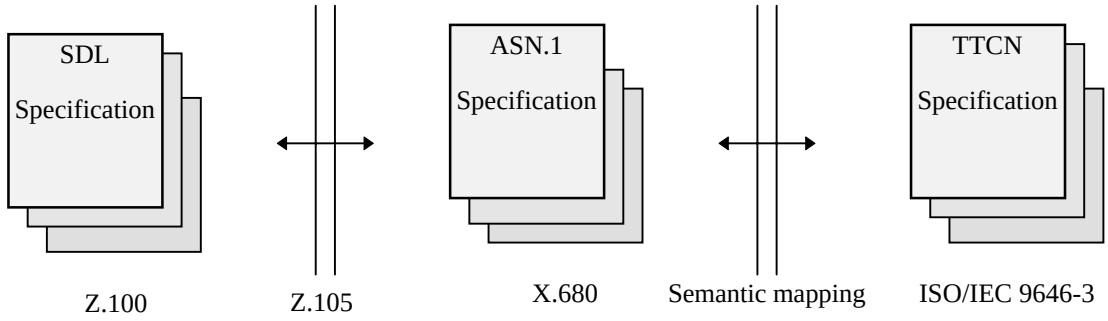


Figure 1: Language interfaces

In other words, the harmonisation and integration paradigm is: all inter-mixing of syntax between SDL and ASN.1 and between TTCN and ASN.1 is removed, missing functionality is added and incompatible functionality is corrected. All necessary changes to these languages is done in the relevant standards. For example, changes necessary for ASN.1 to harmonise with TTCN are made directly to the TTCN standard or to the ASN.1 standard and not, as is the case today, either only from the TTCN standard (=> an ASN.1 dialect) or not at all.

For users this will mean that

- the SDL parts of a specification will be written using **only** the language defined in Z.100 [5];
- the TTCN parts of a specification will be written using **only** the language defined in ISO/IEC 9646-3 [7];
- the ASN.1 parts of a specification will be written using **only** the language defined in the ITU-T X.680 series [1].

The advantages to both users and tools manufacturers of this approach are clear - users will not have to learn special dialects of each language and tool-makers will not need to implement special compilers etc. for each dialect.

5 Changes to SDL

This clause details the changes necessary to harmonise SDL and ASN.1. Annex A of this Technical Report contains the related proposals submitted to ITU-T Study Group 10/Q6.

5.1 Summary of SDL issues

The following list summarises the issues associated with harmonising SDL-ASN.1:

- SDL case-sensitivity;
- removal of in-line ASN.1 from SDL;
- support for encoding rules and tags in SDL;
- support of information objects (classes & sets) and reference to information objects from SDL;
- support of open types in SDL (e.g. constrained by information objects);
- support of ASN.1 parameterisation in SDL;
- ASN.1-SDL type equivalence (i.e., mapping of ASN.1 types to SDL types, including the new ASN.1 string types);
- support of SDL operators for ASN.1 types;
- SDL support for CHOICE, OPTIONAL and DEFAULT constructs.
- SDL support for ASN.1 extensibility.

5.2 SDL case-sensitivity

SDL is currently defined as being case-insensitive as follows (Z.100 [5], p16):

"In all <lexical unit>s except <character string>, <letter>s are always treated as if uppercase. (The treatment of <national>s may be defined by national standardisation bodies.)"

This causes many problems when using SDL with ASN.1, TTCN, C, C++, Java and other programming languages.

Examples of such problems include:

- using or referencing ASN.1 within SDL
- code generation from SDL, where the case sensitivity of identifiers, types, etc. are relevant

Proposed SDL change 1: Change Z.100 to make SDL case sensitive. It is further proposed to define a definite case for keywords (either upper- or lower case).

The consequences of making SDL case-sensitive would require defining case for all keywords, since it would not be of benefit to define parts of the language as case insensitive. Since the current definition defines <letter>s to be treated in uppercase, it is proposed all SDL keywords are defined as uppercase. This also has the advantage of distinguishing keywords easily within SDL specifications. This would change the definition to:

"In all <lexical unit>s <letter>s are case sensitive. <keyword>s are defined in upper case."

All keywords currently defined in Z.100 [5] would be in upper case, instead of lower case as currently written in Z.100 [5].

Because case can be mixed in case-insensitive SDL throughout the specifications, identifiers and names must be defined consistently in case-sensitive SDL. Migration from case insensitive to case sensitive can be done through a series of rules. The basic requirements for migration of existing systems are:

- translate all <keyword>s into upper case
- for <character string>s and <note>s, no translation
- where ASN.1 is referenced, case shall be determined by the case of the ASN.1 definition, and this case used in all references and subsequent definitions
- for all other <word>s within SDL, case shall be determined by the case of definition, and this case used in all references and subsequent definitions

5.3 Removal of In-Line ASN.1 in SDL

Use of in-line ASN.1 in SDL causes many problems (some of which are identified in Z.105 [6]).

Removing in-line ASN.1 from SDL, will provide the following benefits:

- provide a cleaner interface between SDL and ASN.1 in accordance with the proposed paradigm
- remove many syntactical restrictions

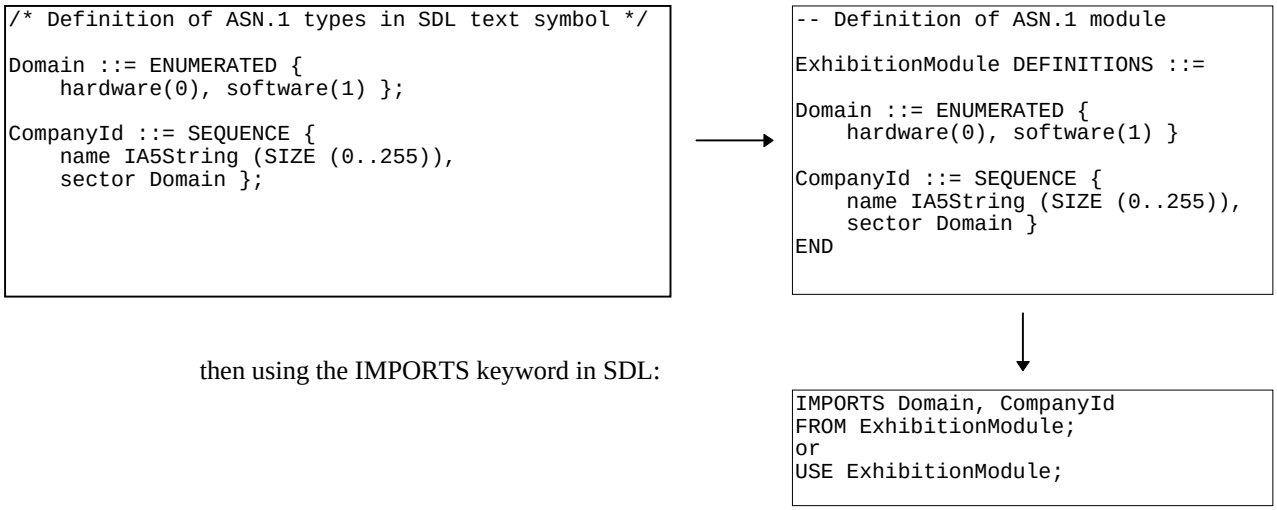
Proposed SDL change 2: Remove in-line ASN.1 from Z.105 (and thus from SDL).

The Consequences of removing in-line ASN.1 are

- The removal of in-line ASN.1 will require the implementation of CHOICE, OPTIONAL and DEFAULT constructs within the SDL core language.
- ASN.1 definitions will be specified in a separate ASN.1 module using only the defined ASN.1 language from the ASN.1:1997 standards.
- No ASN.1 definitions can explicitly appear in-line in the 'SDL part' of a specification.
- ASN.1 types and values are accessible in the 'SDL part' by use of references.
- Migration and backward compatibility are defined as part of this proposal.

Migration would require removing in-line ASN.1 from SDL specifications, and placing them in a separate ASN.1 module. The ASN.1 module corresponds to the SDL package i.e. it will contain type and value definitions. Note the following cases:

- Where ASN.1 types are defined in isolation (no mix of SDL and ASN.1) these can be defined in an ASN.1 module and referenced.



- Where SDL sorts are used in ASN.1 type definitions, these will have to be translated into pure SDL type definitions, with the relevant ASN.1 constructs defined as new type definitions in an ASN.1 module and referenced:

```
/* SDL excerpt */
NEWTYPE PhoneFax
    SEQUENCE OF INTEGER
ENDNEWTYPE PhoneFax;

/* and now the following inline ASN.1
type */
CompanyId ::= SEQUENCE {
    name IA5String (SIZE (0..255)),
contact PhoneFax };
```

→

```
-- ASN.1 module
ExhibitionModule DEFINITIONS ::=

PhoneFax ::= SEQUENCE OF INTEGER

CompanyId ::= SEQUENCE {
    name IA5String (SIZE (0..255)),
contact PhoneFax }
END
```

then using the IMPORTS keyword in SDL:

↓

```
IMPORTS PhoneFax, CompanyId
FROM ExhibitionModule;
```

- Where ASN.1 is used in SDL types, these can be defined in an ASN.1 module and referenced:

In ASN.1, if we have:

```
ListOfDepts ::= SEQUENCE SIZE(2)  OF INTEGER
```

that is used in SDL as:

```
NEWTYPE CompanyCity
nn ListOfDepts,
city IA5String
ENDNEWTYPE CompanyCity;
```

becomes in ASN.1:

```
ListOfDepts ::= SEQUENCE SIZE(2)  OF INTEGER

CompanyCity ::= SEQUENCE {
        nn ListOfDepts,
        city IA5String }
```

that is referenced in the SDL:

```
IMPORTS ListOfDepts, CompanyCity FROM Asn1Module;
```

5.4 Support for encoding rules and tags in SDL

SDL/Z.105 does not allow the definition of encoding rules. A protocol specified in ASN.1 for which encoding rules are given, needs to have a corresponding mapping in SDL, if a code generator is to work directly from SDL source alone.

The other associated problem is with ASN.1 tags. These are used in ASN.1 to distinguish types for encoding computation, they are currently ignored in SDL.

To include the definition of encoding/decoding rules within SDL language would be a complicated and potentially unpopular task. It seems a better solution to have a reference mechanism to encoding rules i.e. an optional field in the SDL specification. This approach is also in line with the TTCN language specification where there is a reference to required encoding rule.

Tags do not necessarily have to be imported into SDL, if ASN.1 is not defined explicitly in SDL. Since tags are only required for encoding / decoding, they can be stored within the code generator or imported at code generation time.

Proposed SDL change 3: Extend SDL to allow optional encoding reference on signal parameters.

The proposal is split into two parts. The first part allows the definition of the encoding rules available. The second part allows reference to the required encoding rule for a particular signal parameter.

Defining the encoding rules available provides a simple way to perform syntax checking in the referencing, preventing nasty spelling errors in the code. To do this we introduce a new construction in SDL.

```
<encoding>      ::= ENCODING ( <encoding rules list> )
<encoding rules list>   ::=     <encoding rule entry> [ ‘,’ <encoding rule entry> ]*
<encoding rule entry>   ::=     <encoding rule name>  [ ‘{‘ <encoding rule variants> ‘}’ ]
<encoding rules variants>::=    <encoding variant name> [ ‘,’ <encoding variant name> ]*
```

For example:

```
ENCODING (PER, BER{definititeLengthEncoding, indefinititeLengthEncoding}, MyEncoding);
```

This example shows how a list of allowed encoding rules can be defined for a system. In addition the construct allow definition of the available encoding variations for each base encoding rule. Now the encoding rules and variations defined for the system can be referenced in the signal definitions. This way we can associate each of the signal definitions with the appropriate encoding rules.

The encoding rules must be specified for each signal parameter individually. The rationale for this is twofold.

1) There could be more than one encoding rule defined for a protocol.

2) In a stack oriented system only signal parameters representing PDU data sent to the environment should be encoded.

The grammar for the signal parameters is changed to:

```
<signal definition item> ::=
        <signal name>
        [<formal context parameters>]
        [<specialization>]
        [<signal parameter list>][<signal refinement>]
<signal parameter list> ::=
        (<signal parameter> {, <signal parameter>}*)
<signal parameter>::=   <sort> [ ( ENCODED BY <encoding rule name> [<variant>]) ]
<variant> ::= VARIANT <encoding variant name>
```

An Example of using the encoding rules for a signal parameter is.

```
SIGNAL NDataReq(
  NAddrType,
  NAddrType,
  NPDUData   (ENCODED BY BER ), /* Parameter Encoding */
  prioType);
```

NOTE- Although the purpose in SDL with encoding rules is not to test those rules the concepts of “encoding variations” is included because it may be important in cases where it is required to automatically generate TTCN from SDL.

A more general solution which might be explored in the future would be to extend ASN.1 by defining a kind of transfer syntax (TSN.1) i.e. a formal definition, that will be part of the ASN.1 BNF to handle the encoding rules in a unique place in the system specification, and that could be used by both SDL and TTCN environments. Alternatively, we could define TSN.1 separately, and just reference the ASN.1. This would give the same flexibility as defining it in ASN.1, but means the encoding is separate. Benefits are that we can then change encoding rules simply by changing the TSN module, the ASN.1 module would remain the same.

5.5 Information Objects in SDL

Information objects provide a generic table mechanism within the ASN.1 language. Such a generic table defines the association of specific sets of field values or types. This feature replaces the earlier MACRO construct (available in ASN.1:1990) and is primarily used to fill-in gaps in a type definition dependant on one or more key fields.

The proposal for support of information objects in SDL recommends basic support for referencing ASN.1 types which use Information Object in their definitions. The proposal defines some restriction on the types of Information Objects which can be referenced, and defines a potential mapping to SDL types for this restricted set. Since the proposal only allows use of a subset

of possible information object types it should be considered as only a partial solution, a full solution will require the resolution of the open issues associated with the new SDL data model being introduced into SDL-2000.

Proposed SDL change 4: Extend Z.105 mapping to allow the instantiation and use of ASN.1 ‘class derived’ types

The proposal is based on the following principles:

- The specification of Information Object Classes, Object class Instantiation and Information object sets should only be possible within an ASN.1 module (these are ASN.1 concepts for describing syntax aspects and have no place in SDL).
- The specification of ASN.1 types which use Information Object Classes and Information object sets in their definitions ('class derived’ types) is also restricted to the ASN.1 module.
- It is through these 'class derived' type that the functionality of the information objects is used, therefore it must be possible to instantiate and use such types within SDL.
- It should not be possible to reference Unconstrained open types from SDL.
- To enable SDL to use such 'class derived’ types a mapping between the elements possible in such an ASN.1 type and SDL must be defined.

The important possible elements in a ASN.1 class related type are shown below

1. normal ASN.1 value field
2. object class fixed type value field
3. object class fixed type value set field
4. object class open type field
5. object class type field with table constraint
6. object class type field with relational constraint

```
ExamplePDU ::= SEQUENCE {
        value       BYTE,
        fixedValue  MSG.&msgCode,
        fixedValues MSG.&Msgs,
        openType    MSG.&Typefield,
        type1       MSG.&Typefield
        ({MsgObjectSet}),
        type2       MSG.&Typefield
        ({MsgObjectSet}{@.fixedValue})}
```

1) The first field type simple since the mapping of normal ASN.1 value fields to SDL is already defined in Z.105 [6].

2) A fixed type value field derived from an information object class is also straight forward to translate, since in principle without constraints it is analogous to field type (1) and can be translated in the same manner by considering the base type defined in the class definition.

3) A fixed type value set field can also be directly translated to SDL by use of Z.105 [6].

4) An open type field derived from an information object class and having no associated constraints (to limit the range of possible types) is not supported in this proposed solution, as stated in the proposal principles.

5) An open type field derived from an information object and having an associated table constraint referencing an information object set can be translated to a corresponding CHOICE type and therefore translated to SDL using the existing transformation rules in Z.105 [6]. An example of this transformation path is shown in section 5.5.1 support for defined Useful Information object types. It should be noted that this transformation does not preserve type equivalence and depending on the encoding rules used may result in a different bit representation on the line.

6) This proposal contains no solution for mapping the relational information contained in an ASN.1 relational constraint to SDL. Therefore an open type field with an associated relational constraint will be transformed as if the relational

constraint were not present (i.e., if a table constraint is present it will be handled as in case (5) otherwise it will be handled as defined for case 4)).

5.5.1 Support for Defined Useful Information Object Types

Suitable mappings for TYPE-IDENITFIER, ABSTRACT-SYNTAX and INSTANCE OF must be defined, where these are declared in ASN.1 and suitably constrained.

The **TYPE-IDENTIFIER** information object class is defined in X.681(Annex A) [2] as:

```
TYPE-IDENTIFIER ::= CLASS
                {
                    &id OBJECT IDENTIFIER UNIQUE,
                    &Type
                }
                WITH SYNTAX { &Type IDENTIFIED BY &id}
```

where there is a &Type field which defines the ASN.1 type for carrying all information about a particular object in the class.

The ABSTRACT-SYNTAX information class is defined in X.681(Annex B) [2] as:

```
ABSTRACT-SYNTAX ::= CLASS
                {
                    &id OBJECT IDENTIFIER UNIQUE,
                    &Type,
                    &property BIT STRING {handles-invalid-encodings(0)} DEFAULT {}
                }
                WITH SYNTAX {
                &Type IDENTIFIED BY &id [HAS PROPERTY &property]
                }
```

where the &id field of each ABSTRACT-SYNTAX is the abstract syntax name, while the &Type field contains the single ASN.1 type whose values make up the abstract syntax and the property “handles-invalid-encodings” denotes that the invalid encodings are not to be treated as an error during the decoding process.

Mapping these information object types to SDL could be based on the Z.105 (page 42) [6] workaround proposal:

For example, let’s consider a process having several attributes, possibly of different types, each attribute being addressed by its unique identifier. The question is how to specify a general operation to read an attribute? The type of the response of the read operation depends on the attribute that was read (see example below); however as the value of the unknown type can only be determined at run time, it is then impossible to derive an equivalent SDL construct based only on this specification.

For example:

```
RESULT-CLASS                ::= TYPE-IDENTIFIER

Result                      ::= OCTET STRING

attribute OBJECT IDENTIFIER ::= { etsi 17}

readResult  RESULT-CLASS    ::={Result  IDENTIFIED BY attribute}
```

could be mapped to an associated type:

```
ReadResultType              ::= SEQUENCE
                                {
                                    attribute   OBJECT IDENTIFIER,
                                    result      OCTET STRING
                                }
```

To make the specification of the read operation possible, we must know the different attribute types. In that case, the TYPE IDENTIFIER construct can be replaced by a CHOICE type over the different attributes. For example, if there are two attributes of respective types INTEGER and BOOLEAN, an intermediate type can then be defined that allows to specify the complete read operation in SDL.

```
ResultType ::=  CHOICE {
                attr1       INTEGER,
                attr2       BOOLEAN };

ReadResult ::=  SEQUENCE {
                attribute   OBJECT IDENTIFIER,
                result      ResultType };
```

In practice we have therefore specified all the possible types for the different identifiers (i.e. attributes) in the class. Such exhaustive description necessary for the mapping may be seen as a potential limitation.

The **INSTANCE OF** type notation is defined in X.681(Annex C) [2] as:

```
InstanceOfType  ::=  INSTANCE OF DefinedObjectClass
```

An INSTANCE OF type is used to specify a type containing an OBJECT IDENTIFIER field and an open type whose value is of a type determined by the OBJECT IDENTIFIER. More precisely: The INSTANCE OF type is restricted to carrying a value from the information object class TYPE-IDENTIFIER. Each INSTANCE OF type has an associated sequence type used to define its values and subtypes, that is described as follows:

```
SEQUENCE
        {
            type_id     <DefinedObjectClass>.&id,
            value       [0] <DefinedObjectClass>.&Type
        }
```

where DefinedObjectClass is the one used in the InstanceOfType definition. For example:

```
ACCESS-CONTROL-CLASS ::= TYPE-IDENTIFIER

Get-Object ::= SEQUENCE {
    objectClass     ObjectClass,
    objectInstance  ObjectInstance,
    accessControl   INSTANCE OF ACCESS-CONTROL-CLASS    --  to be constrained here
}
```

Get-Object is equivalent to:

```
Get-Object ::= SEQUENCE {
    objectClass     ObjectClass,
    objectInstance  ObjectInstance,
    accessControl   [UNIVERSAL 8] IMPLICIT SEQUENCE {
        type-id     ACCESS-CONTROL-CLASS.&id,       --  to be constrained here
        value       [0] ACCESS-CONTROL-CLASS.&Type  --  to be constrained here
    }
}
```

The INSTANCE OF type is useful when it is constrained by an information object set. Therefore, we can only support in SDL such constrained INSTANCE OF types. It could be mapped to SDL in the same way it is proposed for TYPE-IDENTIFIER, i.e. using a CHOICE type to describe all the possible values of the Type field.

5.7 ASN.1 parameterization in SDL

All ASN.1:1997 concepts (even information object classes can be parameterized. This feature allows the partial specification of types or values within an ASN.1 module with the specification being completed by the addition of the actual parameters at Instantiation time.

The ability to specifying parameterised types and values within ASN.1:1997 requires either the definition of transformation rules to allow the use of such entities within SDL or the definition of restrictions to the use of the languages which removes this problem.

In this clause the first solution proposed is to resolve all parameters within the ASN.1 so that the existing mapping to SDL corresponding types (without parameters) can be used. The second solution proposed is to transform ASN.1 parameterized types into SDL data types with formal context parameters.

5.7.1 Solution A for ASN.1 parameterization in SDL

Proposed SDL change 5A: Import only fully defined types and values from ASN.1 modules

The user must resolve any parameterization of ASN.1 types and values before they can be used in SDL. This could be done by a transformation from X.683 [3] to X.680 [1] basic ASN.1 (taking care of the compatibility to Z.105 [6]). This transformation may be done manually or with a tool (some restrictions on the ASN.1 could be imposed).

In order to facilitate this solution, it is necessary to provide some guidance to the users, with recommendations on defining parameterized ASN.1 types for further use in SDL. In addition transformation of ASN.1 parameterized type definitions will remain a tedious task, and this solution may not allow direct use of the original ASN.1 modules in SDL.

For example:

a) Original specification (parameterized ASN.1 in AUTOMATIC Tagging environment)

```
ModuleA DEFINITIONS ::=
BEGIN
-- definition of a parameterized type
ProtocolMsgType { ElementTypeParam } ::= SEQUENCE {
    id      ElementTypeParam,
    num     INTEGER,
    data    IA5String
}
-- definition of a parameterized value
genericString { IA5String : name } IA5String ::= { "Name : ", name}

END
```

b) Intermediate specification (instantiated ASN.1)

To make use of the parameterized types and values defined by *ModuleA*, an intermediate module or extra type declarations may have to be defined as follows:

```
ModuleB DEFINITIONS ::=
BEGIN
    IMPORTS
        ProtocolMsgType, genericString
    FROM ModuleA;

    IdPDUType ::= ENUMERATED {cr, cc, dr, dt, ak}       /* for example */

    DataPDUType ::= ProtocolMsgType { IdPDUType }

    johnGreeting IA5String ::= genericString { "John" }
END
```

c) Using ASN.1 types in the SDL

```
The SDL specification could then be:

use moduleB;
...
process myproc;
    dcl
        ackmsg      DataPDUType,
        seqnumb     Integer,
        mystring    IA5String;

    start;
    ...
    task ackmsg     := ( AK,  2, "Message delivered correctly" );
    task seqnumb    := ackmsg!num;
    task mystring   := johnGreeting;
    ...
    nextstate s1;

    state s1;
    provided true;
    stop;
endprocess myproc;
```

In this solution, very few changes are required in Z.105 [6]. But the SDL user has to expand out any parameterization possibly by defining an intermediate ASN.1 modules as shown above. The translation/instantiation of any ASN.1 parameterized definition may also be rather complex to automate, in particular when information objects are involved.

5.7.2 Solution B for ASN.1 parameterization in SDL

Proposed SDL change 5B: Use formal context parameters in SDL data types

The proposed solution is to translate the ASN.1 parameterized type into an SDL data type with formal context parameters that will preserve the ability to partially specify types.

Formal context parameters have been introduced in SDL to parameterize type specifications (data types, but also process types, signal types).This form of type parameterization in SDL seems capable of providing far greater flexibility in the models, allowing for example straightforward customization for a specific context. Moreover, this SDL feature does not prevent static type-checking at compile time, at the expense of more complexity in the compiler.

5.7.2.1 Formal Context Parameters in SDL type definitions

In the most simple case (i.e. with no parameterization), a type specification is dependent on the context of where it is defined. To define type specifications independent of their context, SDL allows the user to parameterize type specifications with <formal context parameters>.

Parameters can be specified for system type, block type, process type, service type, procedure, signal type and data type. The parameters of a type specification can be the identifier of a process, procedure, signal, variable, timer, synonym or sort (data type). Those parameters are bound either when the parameterized type is instantiated or when the parameterized type is specialized in a subtype definition.

The heading of a parameterized type has the following syntax :

< <formal context parameters> [; <formal context parameters> ; …] >

Each formal context parameter (see example below) starts with a keyword giving the kind of parameter (i.e., process, procedure, signal, variable, timer, synonym, sort) followed by an identifier, and eventually by a constraint that will apply to the actual context parameter.

Process as context parameter

syntax : ***process*** *<process name> [<process_constraint>]* where the constraint can be:

either (1) *[* ***atleast*** *] <process_type name>* or (2) ***fpar*** *<sort> [, <sort> …]*

Format (1) constrains the actual parameter to be of certain type or subtype (if **atleast** is used). Format (2) constrains the actual parameters to conform exactly to the list of sorts given in **fpar**. For example

```
process type P2 <process P1 fpar Integer>;
```

In this example, the process type P2 is parameterized with a process context parameter P1, which is itself constrained to be a process with a formal parameter of type Integer. P1 can then be used in a statement. A process can be a context parameter of the specification of a process type, a service type and a procedure. For example:

```
process type P2 <signal S1 atleast Setup,  S0(Integer)>;
```

In this example, the process type P2 is parameterized with 2 signals S1 and S0. Both signals are constrained: S1 is signal Setup or a subtype of Setup, and S0 is a signal carrying one parameter of sort Integer.

Other possible context parameters are: **variables, timers, synonyms** and **sorts (data types)**.

5.7.2.2 Use of Formal Context Parameters in SDL data types

Considering the ASN.1 specification of *GenericMsg { MsgDataType }* given in section 5.7.1, if we want to use it in SDL, we could do as follows:

```
process myproc;

newtype GenericMsg <newtype MsgDataType endnewtype>
struct
    msgCode :   INTEGER;
    msgLength:  INTEGER;
    msgData :   MsgDataType;
operators
    makeMsg :   INTEGER,INTEGER,MsgDataType ->  GenericMsg;
operator makeMsg;
    fpar    x : INTEGER, y : INTEGER, z : MsgDataType;
    returns GenericMsg;
    dcl
        result GenericMsg;
    start;
        task result!msgCode   := x;
        task result!msgLength := y;
        task result!msgData   := z;
    return result;
```

```
endoperator;
endnewtype GenericMsg;

newtype Message inherits GenericMsg<OCTETSTRING>
endnewtype;

dcl
    ackmsg      OCTETSTRING,
    code        Integer,
    mystring    Message;

start;
...
task mystring   := makeMsg(7,11,'01237');
task ackmsg     := mystring!msgData;
task code       := mystring!msgCode;
...
nextstate s1;

state s1;
provided true;
stop;

endprocess myproc;
```

consequences of using context parameters:

- provide the possibility to directly use any parameterized type definitions from pre-defined ASN.1 modules. In addition formal context parameters will preserve the semantics of the ASN.1 referenced parameterized types.
- changes to the SDL syntax should be limited and the mapping to SDL formal context parameters should be easy to implement in a tool. Note also the possibility to introduce default operators such as the “make” one.
- the use of formal SDL context parameters for ASN.1 parameterization applies only to the type notation. It should be extended for SDL synonyms, in order to deal with the value notation
- by using the inherits facility, it is easy for the user to add new operators and literals, thus providing an alternative for defining SDL operators for ASN.1 types.
- this solution should preserve the resolution of all types at compile time. Therefore SDL compilers can always fully check a specification statically.

5.7.3 Parameterization issues for further investigation

The preferred solution for supporting ASN.1 parameterized types and values in SDL is solution B, however

the following points need be clarified:

- the presence of several levels of parameterization in an ASN.1 type;
- the processing of information objects in SDL (addressed in a separate section) has to be coherent with what is proposed here because these information objects can also be parameterized;
- regarding the SDL use of formal context parameters, it must be verified if more than one type and/or value can be passed as parameters to a SDL data type defined with formal context parameters;
- the possibility of direct mapping from ASN.1 '97 parameterized types to the new data model being developed for SDL 2000.
- the real requirements of users in this area.

In conclusion, the possibility to import ASN.1 parameterized types into SDL would make easier the writing of partial specifications, and the reuse of existing specifications. However up to now, it seems that very few users have been producing ASN.1 specifications in conjunction with SDL, and so it is difficult to find real examples. The final decision on ASN.1 parameterized types in SDL should be deferred until the above points are clarified.

5.8 Extensibility support in SDL

ASN.1:1997 supports the specification of extensible data types by use of the extension marker. Extensible data types allow the definition of different versions of a given protocol which still preserve backwards and forwards compatibility.

Proposed SDL change 6: Extend Z.105 transformation rules to support ASN.1 extensible types

At present extensibility is not supported in SDL. Since the extension marker is directly visible in some transfer syntax's (e.g. PER), simply removing it from the relevant ASN.1 definitions is not an acceptable solution. There is no requirement for adding extensibility or the extension marker directly to the SDL language, therefore the solution proposed is to define a translation as part of Z.105 [6] which encapsulates the following rules:

- types containing extension markers should be translated to SDL in accordance with the normal Z.105 [6] procedures, in addition the following rules must be implemented.
- fields defined after an ASN.1 extension marker will be optional within SDL
- for all fields explicitly defined after an extension marker the operators Make, Present, Modify and Extract will be defined, as currently defined in Z.105 [6] for **choice**
- the SDL entity resulting from the translation of an ASN.1 extensible type must contain one or more extra field to act as a hidden place holder for data elements which are not explicitly defined in the syntax (i.e. from a later version of the protocol). These extra field directly correspond in number and location to the extension marker in the ASN.1 definition. There are no operators defined for these fields it is purely to allow assignment transparency (see example for clarification).

Example:

The ASN.1type definition of Xv11 and Xv12 provide an example of extensibility. If two peer entities are communicating with each other, one using protocol version 1.1 and the other using protocol version 1.2 (as shown in figure 2), interworking is supported at the ASN.1 syntactic level.

Protocol version 1.1	Protocol version 1.2

Xv11::= [42] SEQUENCE{
f1 [1] INTEGER,
…,
f2 [2] INTEGER OPTIONAL
}
	Xv12 ::= [42] SEQUENCE {
	f1 [1] INTEGER,
	…,
	f2 [2] INTEGER OPTIONAL,
	f3 [3] INTEGER OPTIONAL
	}

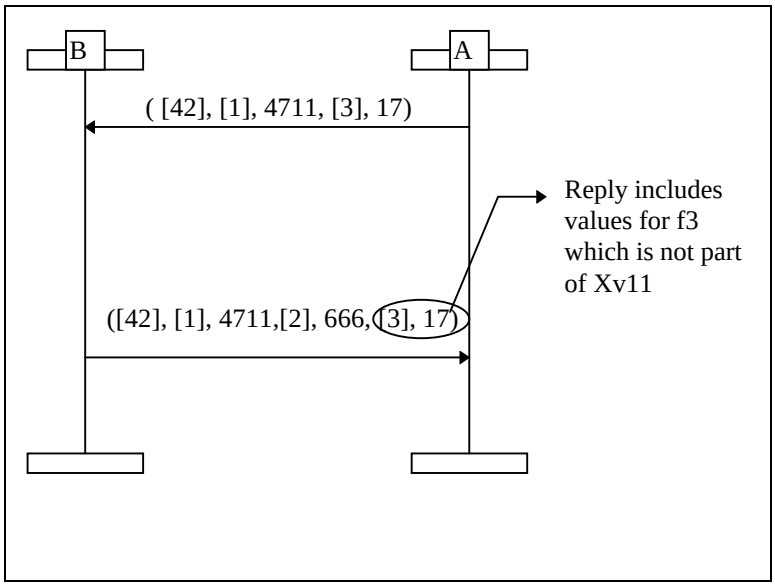


Figure 2: Extensibility example

To clarify the need for the extra place holder field in the SDL equivalent of an ASN.1 extensible type, consider the example in figure 2 where process A is using protocol version 1.1 and process B is using protocol version 1.2. When process A sends a message of type Xv12 to process B but omits the value for the optional field f2, this is message is recognised by B who decodes it according to Xv11, storing the undecoded extension in the hidden SDL field. When process B wishes to use this received signal, for example by returning the received signal back to process A, the stored extension can be added to the encoding of the message.

5.8 ASN.1 - SDL Type Equivalence

We need to define a clear interface between ASN.1 and SDL that specifies unambiguously the type correspondence. This will extend Z.105 [6] appropriately in order to manipulate variables of a type defined in ASN.1 from SDL.

ASN.1 provides the subtyping mechanism to limit values of an existing type. All operators of the original type apply to the subtype, but if the result of an operation on a subtype is out of range, the result is undefined so is the resulting behaviour of the system. In SDL, it is always recommended to use subtyping in order to restrict the size of data types by size constraints. ASN.1 subtypes are mapped to SDL syntypes.

To be able to use the ASN.1 String types in an SDL environment it is essential there is succinct and easy way to define string values.

Proposed SDL change 7: Extend the <character string literal> production to allow *"string"*, *' 'B* and *' 'H* value notation within Z.100

The <character string literal> production should be extended with the new string types.

The <character string> production should be kept as it is the base for the comments and informal text.

```
<character string literal> ::=  <character string> | <character string list> | <quadruple> | <tuple> |
                                <bitstring> | <hexstring> | <quoted string>
<character string list>::= { <charsyms> }
<charsyms> ::= <charsdfn> [, <chardfn> ]*
<chardfns> ::= <cstring> |<quadruple> | <tuple> | <defined value>
<quadruple> ::={ <group> , <plane> , <row> , <cell> }
<group> ::= <number>
<plane> ::= <number>
<row>    ::= <number>
<cell> ::= <number>
<tuple> ::= { <table column>, <table row> }
<table column> ::= <number>
<table row> ::= <number>
<quoted string> ::=  " <text> "
```

```
<bitstring> ::= <apostrophe> { 0 |  1  }* <apostrophe> B
<hexstring> ::= <apostrophe> {<decimal digit> |  A  |  B  |  C  |  D  |  E  |  F }*
                <apostrophe>  H
```

Example:

```
DCL special BMPString   := "A char = "// {1,2,3,4} // ".";
DCL address OCTETSTRING := 'FFFF'H;
DCL address BITSTRING   := '1101'B;
```

5.9 Support of SDL operators for ASN.1 types

ASN.1 does not support operators and Z.105 [6] does not give any clean possibility to add them in the SDL specification. Not enabling the use of operators is defying the purpose of encapsulation introduced by object orientation in SDL. This limitation will affect the resulting SDL specification and possibly reduce the usability of ASN.1

The support in Z.105 [6] today is not sufficient as the redefinition of the type (e.g. by inheritance) will create a new type which is not shared with other possible users of the same ASN.1 module. The type must also keep its tag information for encoding reasons. Even though Z.105 [6] does not support tagging in SDL, we have to be able to identify a SDL type as type equivalent to the ASN.1 type. Preserving the name for readability would also be of interest.

Proposed SDL change 8: Extend reference clause (Z.105 import clause) to allow association of SDL operators with ASN.1 definitions

The solution proposed is to extend (associate) operators with ASN.1 types when they are imported / used. This will require changes to Z.100 [5] and Z.105 [6] to support both methods of reference. An example of this is

```
<package reference clause> ::=
        use <package name> [ / <definition selection list> ] <end> |
        use <package name> <type redefinition>* enduse <package name>

<definition selection list> ::=
        <definition selection> {, <definition selection>}*

<definition selection> ::=
        [<entity kind>] <name> [ <type redefinition> ]

<entity kind> ::=
        system type
    |   block type
    |   process type
    |   service type
    |   signal
    |   procedure
    |   newtype
    |   signallist
    |   generator
    |   synonym
    |   remote

<type redefinition> ::= type <type name> adding operators <operator definitions>
    endtype [<type name>]
```

This is an example of how operators could be added to imported ASN.1-types

First the ASN.1 Module from which the definitions are imported.

```
MODULE MyModule ::=
BEGIN
Expert ::= SEQUENCE
    {
      name      NameType,
      hotel     HotelType,
      car       RentalCarType,
      hunger    INTEGER OPTIONAL,
      task      TaskType    OPTIONAL
    }

STF ::= SEQUENCE OF Expert

stf121 STF := {
              { "Linus", "Ambasador", ""},
              { "Ian",   "Ambasador", "VWPolo"}
}
```

```
NameType ::= BMPString

HotelType ::= BMPString

RentalCarType ::= BMPString

TaskType ::= ENUMERATED{editor, …}
END MyModule;
```

Then the SDL Use Clause will be used to import the declarations into the SDL system using the transformation rules from Z.105 [6]. There are two ways to import an ASN.1 type and define operators for it depending on whether the use statement globally imports all types from a module or explicitly imports the type we wish to add operators to.

First consider the case when the use construct imports all types from a module. In this case a type redefinition is allowed to add operators.

```
Use myModule
Type Expert
adding
    operators
      assign:   Expert, Work -> Expert
      feed:     Expert, Food -> Expert
      ready:        Expert -> Boolean
EndType Expert;

EndUse MyModule;
```

In the second alternative where you specify each type you want to use respectively you can add the operators at the same time.

```
Use myModule/Expert
adding
    operators
      assign:   Expert, Work -> Expert
      feed: Expert, Food -> Expert
      ready:    Expert -Boolean
EndType Expert;


Use myModule/STF;

Use myModule/stf121;
```

Finally, an example of referencing the types and the operators.

```
DCL     Linus, HappyLinus, MrHyde Expert;
DCL  destination HotelType;

TASK HappyLinus := feed(Linus, Pizza);

TASK MrHyde := feed (stf121(1), Beer );

TASK destination := stf121(1)!hotel;
```

The extension should be limited to importing of ASN.1 types only and not be general to SDL. The rationale for this is that the solution is extending the use of ASN.1 with operators because they cannot be defined as part of ASN.1. Making extensions applicable to SDL types also *is* creating a new type, and inheritance should be used.

5.9.1 Inheritance of ASN.1 data types

It should be possible to inherit **import**ed / **use**d ASN.1 data types in the same manner as SDL sorts, and be able to add literals and operators to the new type definition. This would ensure that the only difference in usage between SDL sorts and ASN.1 type definitions is in the **use** clause, and the rationale for this difference has already been stated. Any operators associated with an ASN.1 definition when first referenced would also be available in any inherited types.

5.9.2 Multiple references to packages / definitions

It will often be the case that whole packages are imported / used into an SDL specification, but that only some of the definitions require operators associating with them. Given the SDL definition:

```
Use myModule;
Use myModule/Expert;
```

this is currently allowed as defined in Z.100 [5] (the union of reference clauses is taken as the references). However, given the definitions:

```
use myModule/Expert
adding
  operators
    feed: Expert, food -> Expert;
endnewtype Expert;

Use myModule/Expert
adding
  operators
    feed: Expert, Beer -> Expert;
endnewtype Expert;
```

two operators are then defined with different signatures for the same ASN.1 definition. The proposed solution to this is that ASN.1 referenced definitions can only have operators associated with them once, when first referenced, within a defined scope. Since SDL allows package reference clauses not only at many scope levels, this allows that an ASN.1 definition can have different operators associated with it if referenced within two separate scope units.

5.10 SDL support for CHOICE, OPTIONAL and DEFAULT

This item is being addressed by ITU-T SG10-Q 6 group.

5.11 List of proposed changes to SDL

No.	Description	Note
1	Change Z.100 [5] to make SDL case sensitive. It is further proposed to define a definite case for keywords (either upper- or lower case).	
2	Remove in-line ASN.1 from Z.105 [6] (and thus from SDL).	
3	Extend SDL to allow optional encoding reference on signal parameters.	
4	Translate ASN.1 Information Objects / Sets to SDL Types	Dependent on SDL-2000 data model
5A	Import only fully defined types and values from ASN.1 modules	
5B	Use formal context parameters in SDL data types	Dependent on SDL-2000 data model
6	Extend Z.105 transformation rules to support ASN.1 extensible types	
7	Extend the <character string literal> production to allow *"string"*, *' 'B* and *' 'H* value notation within Z.100	
8	Extend reference clause (Z.105 import clause) to allow association of SDL operators with ASN.1 definitions	

Table 1: List of changes to SDL

6 Changes to TTCN

6.1 Summary of TTCN issues

The following list summarises the issues associated with harmonising TTCN-ASN.1:

- ASN.1 constraints in TTCN;
- additional ASN.1 string types in TTCN;
- ASN.1 information objects in TTCN;
- ASN.1 type parameterization in TTCN;
- ASN.1 extension markers in TTCN;
- automatic ASN.1 tagging in TTCN;
- ASN.1 exception identifiers in TTCN.

6.2 Support of ASN.1 Constraints in TTCN

This clause is divided into two sections. The first section concerns the specification of values within ASN.1 constraints. The second section describes proposals to allow matching mechanisms to be specified for constraint fields.

6.2.1 Specification of Constraint Values

This section deals with issues of the TTCN / ASN.1 interface associated with specifying a distinct value to an ASN.1 constraint field. As illustrated below:

- The use of TTCN test suite constants in ASN.1 constraints proformas.
- The use of expressions in ASN.1 constraints proformas.
- The inclusion of ASN.1 field identifiers in ASN.1 constraints proformas.

ASN.1 PDU Type Definition
PDU Name : T_CONNECT1 PCO Type : Comments :
Type Definition
-- ASN.1 type definition in TTCN SEQUENCE { source BIT STRING (SIZE (4..4)), destination BIT STRING (SIZE (4..4)), t_Class INTEGER (0..4), userData IA5String OPTIONAL }

Figure 3: Typical ASN.1 PDU Type Definition

ASN.1 PDU Constraint Declaration
Constraint Name : TCON_Class4_1 PDU Type : T_CONNECT1 Derivation Path : Comments :
Constraint Value
{ source TS_PAR1, TS_PAR2, -- field identifier can be omitted if desired t_Class 4, userData "testing, testing" }

Figure 4: ASN.1 PDU Constraint allowed by current TTCN syntax

If we consider the example in figure 5 taken from the TTCN edition 2 specification, TS_PAR1 and TS_PAR2 are TTCN test suite Constants. If we try to represent the ASN.1 PDU constraint declaration from figure 5 literally in ASN.1 value notation, we get the following:

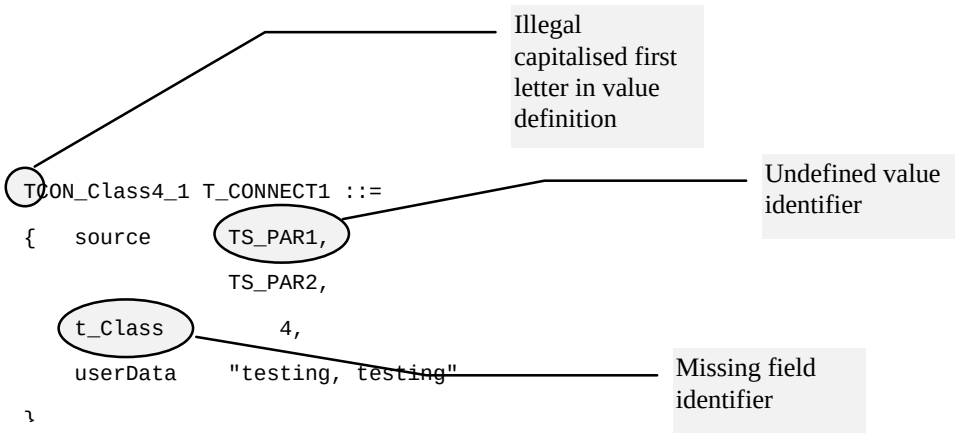


Figure 5: Literal ASN.1 representation of PDU constraint

Using ASN.1 value parameterisation we can define the following, equivalent, legal ASN.1 construct:

In TTCN

```
TCON_Class4_1 ::= constraint1{ TS_PAR1, TS_PAR2 }
```

In ASN.1

```
constraint1{ BIT STRING : tTCN_CONST1, BIT STRING : tTCN_CONST2} T_CONNECT1 ::=
{   source          tTCN_CONST1,
    destination     tTCN_CONST2,
    t_Class         4,
    userData        "testing, testing"
}
```

Figure 6: Proposed representation of an ASN.1 PDU constraint

This transformed version now provides a clean defined interface between the two languages with type checking applied at this interface. From this example the first proposed change is:

Proposed TTCN change 1:	Field identifiers are mandatory in all ASN.1 constraints proformas.

This change makes the TTCN specification consistent with the ASN.1 language definition X.680 [1] in respect to mandatory field identifiers.

The next proposed change is to handle TTCN test suite constants within ASN.1 constraints proformas by using parameterisation. This is consistent with the way TTCN test suite variables are handled within the current version of the TTCN standard.

Proposed TTCN change 2:	TTCN Test Suite Constants must be passed into ASN.1 constraints definitions as actual parameters.

The next proposed change addresses the issue of expressions. The proposed solution is in line with the solution for test suite constants, that is to extract any functionality which is alien to the ASN.1 language from the ASN.1 constraints definitions and insert it within the TTCN.

Proposed TTCN change 3:	The use of expressions within ASN.1 constraints definitions shall be forbidden.

In effect the TTCN user can retain the same functionality by use of parameterisation. Instead of defining the expression within the ASN.1 constraint, the expression would be specified (and evaluated within the TTCN dynamic behaviour and the result passed into the ASN.1 constraint as an actual parameter.

6.2.2 Specification of Matching Attributes

The remaining problem is the use of TTCN value constraint types within the ASN.1 PDU constraints declaration to define matching attributes as shown in figure 6.

ASN.1 PDU Constraint Declaration		
Constraint Name	:	TCON_Class4_2
PDU Type	:	T_CONNECT1
Derivation Path	:	
Comments	:	

Constraint Value
{ source ?, connect ?, t_Class *, userData "Help!" }

Figure 7: ASN.1 PDU constraint declaration using wildcards

The TTCN value constraints (i.e.'?' any value and '*' any or omit) are clearly not part of the ASN.1 language definition. Indeed the whole concept of "matching specification" has little correlation with the specification of abstract syntax's.

6.2.2.1 Proposed Solution: TTCN Table Constraints

In this solution the removal of TTCN syntax from ASN.1 notation is taken to its logical conclusion, all non-standard ASN.1 syntax is removed from the TTCN ASN.1 constraints specification proforma. This solution is directly in line with the overall paradigm of removing non-standard dialects of ASN.1. The matching mechanisms which are at present coerced into ASN.1 by the TTCN redefinition of DefinedValue are removed and the existing functionality of the TTCN Language used instead.

Proposed TTCN change 4: Only the value notation syntax defined within the X.680 [1] standard shall be used within ASN.1 constraints proformas. The redefinition of the ASN.1 production for DefinedValue in the TTCN BNF is deleted.

This solution means that within the ASN.1 constraints proformas TTCN value constraints ('*', '?' and OMIT) would not be allowed. The use of test case variables, test suite variables and test suite constants would still be allowed by use of direct parameterisation.

In effect this will limit the use of the TTCN ASN.1 constraints to defining distinct values for all fields by removing the ability to specify matching mechanisms. To replace this functionality it is proposed that ASN.1 constraints can also be defined within TTCN tabular constraints proformas

Proposed TTCN change 5: Constraints for ASN.1 types can be defined using TTCN tabular constraints

Within the tabular constraints the user can use the full range of TTCN value constraints to define the required matching scheme for ASN.1 types.

This proposal states that given the type defined in figure 6, the TTCN ASN.1 PDU constraints proforma can only be used to specify examples of this type with definite values as shown in figure 7. This proforma therefore becomes most useful in the specification of transmit constraints. An example of a receive constraint for this PDU type is shown in figure 10. This demonstrates the use of an TTCN table constraint to define a constraint for an ASN.1 type.

ASN.1 PDU Type Definition		
PDU Name	:	T_STF121
PCO Type	:	
Comments	:	

Type Definition
SEQUENCE { month INTEGER (1..12), day INTEGER (1..31), problem IA5String OPTIONAL }

Figure 8: Typical of an ASN.1 PDU type

ASN.1 PDU Constraint Declaration		
Constraint Name	:	Tuesday
PDU Type	:	T_STF121
Derivation Path	:	
Comments	:	

Constraint Value

```
{   month       5,
    day         29,
    problem     "Last night (and early the next morning!)"
}
```

Figure 9: Typical ASN.1 PDU Constraint

PDU Constraint Declaration		
Constraint Name	:	Wednesday
PDU Type	:	T_STF121
Comments	:	

Field Name	Field Value	Comments
month	?	
day	14	
problem	*	

Figure 10: Using a tabular constraint for an ASN.1 PDU type

This example shows the proposal for simple ASN.1 PDU types. In order to support more complex ASN.1 types, within TTCN tabular form the type mapping between TTCN and ASN.1 must be extended.

Proposed TTCN change 6: Extend TTCN tabular constraint proforma field name syntax to allow referencing to ASN.1 record structures using the existing RecordRef production.

This proposal extends the field name syntax within the TTCN tabular constraint to allow referencing of particular elements within an ASN.1 SEQUENCE OF or SET OF construct. This feature already exists for TTCN expressions using the RecordRef production.

To illustrate the use of this proposal and to demonstrate the use of tabular constraints for complex ASN.1 types the following section uses an personnel record example taken from Annex F of X.680 [1].

The ASN.1 PDU type proforma defined in figure 11 conforms to the present TTCN language specification. It defines a type which includes a SEQUENCE OF construct. Using this example to illustrate the proposed language changes. The ASN.1 constraint shown in figure 12 is ‘pure’ ASN.1:1997 value notation and could be used for transmit events and base constraints and figure 13 demonstrates the specification of a receive constraint for a complex ASN.1 type using a tabular proforma.

ASN.1 PDU Type Definition		
PDU Name	:	PersonnelRecord
PCO Type	:	
Comments	:	

Type Definition

```
[APPLICATION 0] SET
    {   name            Name,
        title           VisibleString,
        number          EmployeeNumber,
        dateOfHire       Date,
        nameOfSpouse     Name,
        children         SEQUENCE OF ChildInformation DEFAULT {}
    }

    ChildInformation ::= SET
    {   name            Name,
        dateOfBirth     Date
    }

    Name ::= [APPLICATION 1] SEQUENCE
    {   givenName       VisibleString,
        initial         VisibleString,
        familyName      VisibleString
    }

    EmployeeNumber ::= [APPLICATION 2] INTEGER

    Date ::= [APPLICATION 3] VisibleString
```

Figure 11: A more complex ASN.1 PDU Type Definition

ASN.1 PDU Constraint Declaration		
Constraint Name	:	John_P_Smith
PDU Type	:	PersonnelRecord
Derivation Path	:	
Comments	:	

Constraint Value

```
{   name            {givenName "John", initial "P", familyName "Smith"},
    title           "Director",
    number          51,
    dateOfHire      "19710917",
    nameOfSpouse    {givenName "Mary", initial "T", familyName "Smith"},
    children
        {   {   name  {givenName "Ralph", initial "T", familyName "Smith"} ,
                dateOfBirth "19571111"},
            {   name  {givenName "Susan", initial "B", familyName "Jones"} ,
                dateOfBirth "19590717" }
        }
    }
```

Figure 12: ASN.1 PDU Constraint on *PersonnelRecord* (no wildcards)

PDU Constraint Declaration		
Constraint Name	:	AllSmiths
PDU Type	:	PersonnelRecord
Comments	:	

Field Name	Field Value	Comments
name.givenName	?	
name.initial	?	
name.familyName	"Smith"	
title		
number		
dateOfHire		
nameOfSpouse.givenName		
nameOfSpouse.initial		
nameOfSpouse.familyName		
children.[0].name		
children.[0].initial		
children.[0].familyName		
children.[0].dateOfBirth		

Figure 13: Using a tabular PDU Constraint for *PersonnelRecord* (with wildcards)

Currently types defined in the TTCN tabular format have all fields optional by default. In order to harmonise with ASN.1 and also from the requirement to explicitly define properties of the IUT within a conformance testing language it is proposed to change this feature.

Proposed TTCN change 7: Within TTCN tabular type definitions all fields are mandatory unless explicitly defined as optional.

6.3 Additional ASN.1 String Types in TTCN

ASN.1:1994 introduced the string types:

- `BMPString`
- `UniversalString`

And ASN.1:1997 introduces the string type:

- `UTF8String`

In order to handle those new string types in TTCN, it is proposed that they are included in the TTCN language.

Proposed TTCN change 8: Include the new string types: BMPString, UniversalString and UTF8String in TTCN.

6.4 Information Objects in TTCN

Information objects provide a generic table mechanism within the ASN.1 language. This feature replaces the earlier macro notation and is primarily used to fill-in gaps in a type definition dependant on one or more key fields.

There are three operations associated with ASN.1 information objects, Definition, Instantiation and referencing. Information object definitions specify the format of the table, how many columns and what attributes are associated with each field.

After definition one or more information objects can be instantiated into an information object set (conceptually a row in the table for each object) by proving specific values and types for the relevant fields.

After instantiation the fields of the information object can be referenced to provide types and values.

The proposed solution allows all three operations to be specified from within TTCN as defined in the following clauses. This approach is in line with the current TTCN language were all the supported language features can be directly specified in one or other TTCN proforma.

6.4.1 Defining Information Objects

Proposed TTCN change 9:	Add new proforma and BNF productions to allow definition of information objects Classes.

The format of the information object proforma is shown in figure 14. These proformas should be included in the declarations part of the test suite. The proposed new hierarchy for the declarations part is shown below:

Declarations Part

Test Suite Type Definitions

Simple Type Definitions…

Structured Type Definitions…

ASN.1 Type Definitions…

ASN.1 Type Definitions By Reference…

ASN.1 Information Object Class Definitions…

Encoding Rules…

ASN.1 Information Object Class Definition		
Class Name	:	*ASN1_ClassId&FullId*
Group	:	*[ASN1_ClassGroupReference]*
Comments	:	*[FreeText]*
Class Definition		
ASN1_Class		
Detailed Comments:		*[FreeText]*

Figure 14: New proforma for ASN.1 Information Object Class Definition

ASN.1 Information Object Class Definition		
Class Name	:	MESSAGE
Group	:	
Comments	:	Example of information object Class definition
Class Definition		

```
CLASS { &msgCode        INTEGER UNIQUE,
        &msgLength      INTEGER,
        &MsgDataType    OPTIONAL
        }
WITH SYNTAX
{   CODE            &msgCode
    LENGTH          &msgLength
    [DATA TYPE      &MsgDataType]
}
```

Figure 15: Example ASN.1 Information Object Class Definition

6.4.2 Declaring Information Object Sets

Proposed TTCN change 10: Add new proformas and BNF productions to allow declaration of information objects and information object sets.

The proposed proforma for information object declaration is shown in figure 16 with the definition of the information object set proforma in figure 17. These proformas should be placed in the constraints part of the TTCN test suite. The resultant hierarchy is shown below:

Constraints Part

Test Suite Type Constraint Declarations

Structured Type Constraints Declarations…

ASN.1 Type Constraints Declarations…

ASN.1 Information Object Declarations…

ASN.1 Information Object Set Declarations…

ASP Constraints Declarations…

…

ASN.1 Information Object Declaration	
Object Name :	*ObjectId&ParList*
Group :	*[ASN1_ObjectGroupReference]*
Object Class :	*[ObjectClassId]*
Comments :	*[FreeText]*
Object Declaration	
ASN1_Object	
Detailed Comments:	*[FreeText]*

Figure 16: New proforma for ASN.1 Information Object Declaration

ASN.1 Information Object Set Declaration	
Object Set Name :	*ObjectSetId&ParList*
Group :	*[ASN1_ObjectSetGroupReference]*
Object Class :	*[ObjectClassId]*
Comments :	*[FreeText]*
Object Set Declaration	
ASN1_ObjectSet	
Detailed Comments:	*[FreeText]*

Figure 17: New proforma for ASN.1 Information Object Set Declaration

ASN.1 Information Object Declaration	
Object Name :	setup
Object Class :	MESSAGE
Comments :	Example of information object declaration
Object Declaration	

```
{
    CODE    1
    LENGTH  12
    DATA TYPE OCTET STRING
}
```

Figure 18: Example ASN.1 Information Object Declaration

ASN.1 Information Object Set Declaration	
Object Set Name :	ConnectPhaseMsgs
Object Class :	MESSAGE
Comments :	Example of information object set declaration

Object Set Declaration

```
{
    setup | setupAck | release | relAck
}
```

Figure 19: Example ASN.1 Information Object Set Declaration

6.4.3 Referencing Information Objects

The ability to reference types and values from ASN.1 information object from within TTCN is already implicitly included in the existing TTCN language specification. This is because the TTCN BNF specification references the ASN.1 productions *Type* and *Value* which in turn give access to the information objects

6.5 ASN.1 parameterization in TTCN

Current TTCN supports only a part of the possible ASN.1:1997 parameterization features. To allow full and consistent use of ASN.1:1997 parameterization from within TTCN the language must be extended to resolve two major issues. The first issue to resolve is the provision of a unique parameter syntax for TTCN and ASN.1. The second issue is extending the TTCN proformas and syntax to support the value and type parameterization provided by ASN.1 (including generic type parameterization but not open types).

6.5.1 Parameterization syntax

The current TTCN language defines the syntax of a parameter list in the following way:

```
( name1 : TYPE1; name2 : TYPE2 )
```

whereas in ASN.1:1997 takes the form:

```
{ TYPE1 : name1, TYPE2 : name2 }
```

The proposed solution is to adopt the ASN.1 format for parameter syntax because of the enhanced functionality (for type parameterization) and the fact that any ASN.1 module must be written using this format to conform to the X.683 standard.

Proposed TTCN change 11: The TTCN syntax for the parameter lists for all relevant proformas will conform to ASN.1:1997

It should be noted that this solution removes backward compatibility to all existing TTCN test cases and also mean that the TTCN users must change to using the new ASN.1 format in all new test suites. However these changes have simple transformations from existing test suites, and where enhancements are not used, a reverse transformation back to the existing TTCN standard can be used. The primary problem for users of TTCN is that they will be forced to use different notations depending on the TTCN language version used, which may cause confusion. However, since a similar problem already exists when between TTCN and ASN.1, it is proposed that the best solution is to standardise both languages on a common notation which at least will ensure that in the future when tools support the new version of TTCN this problem will not exist.

6.5.2 Proforma changes to support parameterization

Currently, parameterization is defined within TTCN in the following:

- value parameterization in ASP & PDU constraints
- "partial" type parameterization in ASPs, by being able to define PDUs as parameters
- parameterization of test steps

In contrast, ASN.1:1997 provides a much broader and more powerful idea of parameterization which includes the concept of type parameters, value sets and access to information objects.

Proposed TTCN Change 12: Extend TTCN proformas and productions to include ASN.1:1997 parameterization.

To be able to use type and value parameterization in TTCN, it is necessary to change the relevant syntax and proformas to allow type and value parameterization to be defined and used. Specifically, the proformas which will require changes are:

- PDU Type Definition;
- ASP Type Definition;
- Structured Type Definition;
- ASN.1 Type Definition by Reference;
- ASN.1 PDU Type Definition;
- ASN.1 Type Definition;
- ASN.1 ASP Type Definition;
- PDU Constraint Declaration;
- Structured Type Constraint Declaration;
- ASP Constraint Declaration;
- ASN.1 PDU Constraint Declaration;
- ASN.1 Type Constraint Declaration;
- ASN.1 ASP Constraint Declaration;
- Default Dynamic Behaviour;
- Test Step Dynamic Behaviour;
- Test Case Dynamic Behaviour.

All these proformas with the exception of test case dynamic behaviour require extensions to allow the passing in of type and value parameters using the ASN.1:1997 syntax. The test case dynamic behaviour proforma is only extended to allow the actual parameter list to be specified for referenced parameterized constraints, type specification in the behaviour description, default behaviour and test step behaviour.

The support of type parameterization does raise issues associated with type checking. In principle a type or constraint that contains type parameterization represents only a partial specification (some type information is missing until the actual parameters are provided) and cannot be checked for consistency in isolation.

The solution proposed limits this problem by imposing the following rules:

1) No type definition referenced from within a test case dynamic behaviour may contain an open type. This means that any type or constraint actually used in a test case can be fully resolved and type checked.

2) A parameterised type definition (proformas 1-7 above) can receive the associated actual parameters either from another type definition referencing this type or via the type reference of the behaviour description column within a test case behaviour, test step behaviour or default behaviour proforma.

3) A parameterized constraint declaration (proformas 8-13) can only use type parameters to allow the specification for generic value passing (see X.683) the dummy type is only used in the parameter list to specify the type of the value. It can receive the associated actual parameters via a chaining constraint (it which case the parameters must be included in the derivation path), or via a constraints reference within a test case behaviour, test step behaviour or default behaviour proforma.

4) Constraint declarations cannot pass parameters to Parameterized type definitions.

5) The test step behaviour and default behaviour can receive actual parameters from the test case behaviour definition. These proformas support type and value parametrization.

These parameter passing relationships between the proforma types are shown in figure 20. Examples illustrating these rules are given in figures 21 -23

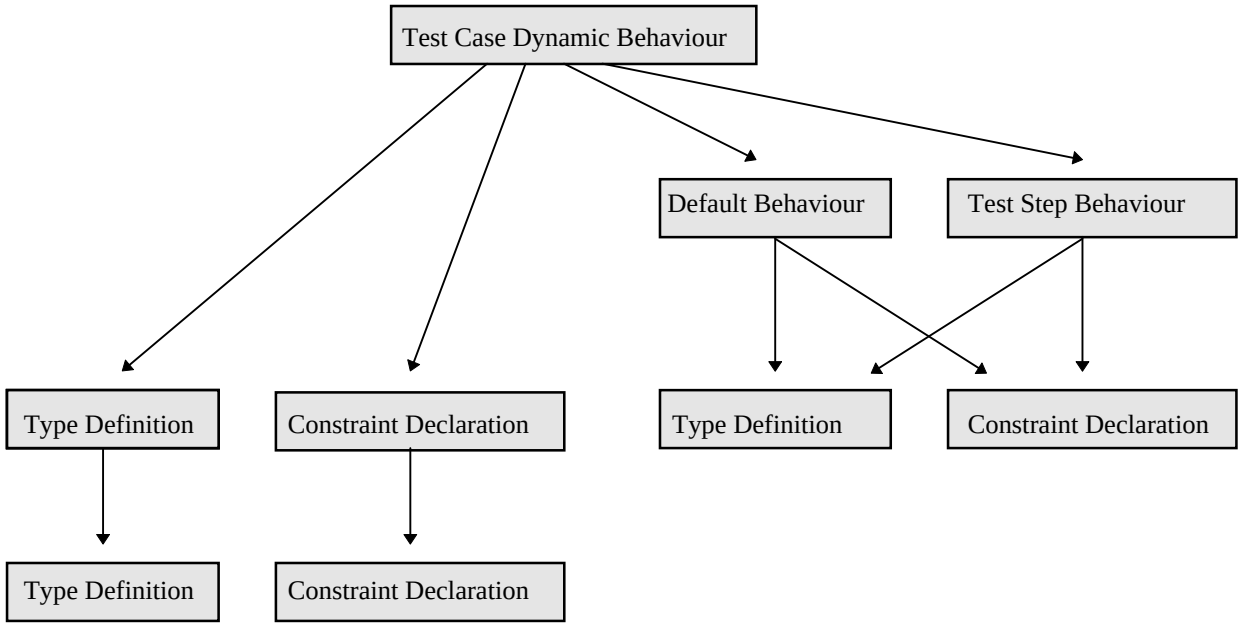


Figure 20: Parameter passing relationships between the proforma types

PDU Type Definition		
PDU Name :	INTC{PTIType, EXTRAType, INTEGER:gfiLength}	
Group :		
PCO Type :	NSAP	
Encoding Rule Name :		
Encoding Variation :		
Comment :		
Field Name	**Field Type**	**Comments**
GFI	BITSTRING [gfiLength]	
LCGN	BITSTRING	
LCN	BITSTRING	
PTI	PTIType	
EXTRA	EXTRAType	

Figure 21: Example of type and value parameterisation in a PDU type definition

ASP Type Definition		
ASP Name :	DataConfirmation	
PCO Type :	NSAP	
Comment :	Parameterized PDU used	
Parameter Name	**Parameter Type**	**Comments**
CallingNetworkAddress	HEXSTRING	
CallingNetworkAddress	HEXSTRING	
ConnectionIdentifier	HEXSTRING	
Data	INTC{OCTET, HEXSTRING, 4}	

Figure 22: Parametrized PDU type receiving actual parameters in ASP Type definition

Test Case Dynamic Behaviour					
Test Case Name : Example1 Group : Purpose : Default : Comments :					
No	**Label**	**Behaviour Description**	**Constraint Ref**	**Verdict**	**Comments**
1 2 3		L! INTC{OCTET, HEXSTRING, 4} L? INTR	ndr_req{ OCTET, octetValue} ndr_rsp{ OCTET, octetValue}	PASS	1 2 3

Figure 23: Parameterised PDU type receiving actual parameters directly from Behaviour Description.

PDU Constraint Declaration		
Constraint Name : ndr_req{ DummyType, DummyType:Value} **PDU Type** : INTC **Derivation Path** : **Comment** :		
Field Name	**Field Value**	**Comments**
GFI	'1000'B	
LCGN	'0001'B	
LCN	'0000'B	
PTI	Value	
EXTRA	*	

Figure 24: Example of parameterised constraint

PDU Constraint Declaration		
Constraint Name : ndr_req1{ DummyType, DummyType:Value} **PDU Type** : INTC **Derivation Path** : ndr_req{ DummyType, DummyType:Value}. **Comment** :		
Field Name	**Field Value**	**Comments**
GFI	'1111'B	

Figure 25: Example of chained parameterised constraints

The macro symbol (<-) used instead of a field / parameter name is equivalent to a macro expansion. Since this can only be used with structure types defined in tabular form, the definition of parameterized ASP / PDU type definitions could cause errors when used.

The proposed solution is to not allow parameterized types which are defined as a macro expansion. This would seem a valid proposal, since it is not clear whether expansion would actually be wanted in these instances where the type is not statically fixed.

Proposed TTCN change 13: Macro expansion in type definitions cannot be performed on parameterized types.

6.6 Extension Markers in TTCN

ASN.1:1997 allows extensibility to be specified within a syntax definition, either explicitly using the extension marker "…" or globally to an ASN.1 module by assigning an optional field in the module header (X.680 [1]). The extension marker is visible in some transfer syntaxes such as PER, and so removing it from ASN.1 to be compatible for reference by TTCN is not an acceptable solution.

Although there is no other strong requirement to add extensibility to TTCN, it should be possible to reference ASN.1 specifications where extension markers are used. The simplest solution is then to allow the extension marker "…" to be used in TTCN ASN.1 syntax.

This would provide the ability to write test suites for an IUT using encoding rules e.g. PER where the extension marker is present, also to allow a direct reuse of existing ASN.1 modules conforming to X.680 Amendment 1 [2], and additionally to be in line with Z.105 [6] which supports extension markers.

The one remaining issue is the semantics of extension markers for constraints operating on an extended type. It is proposed that any constraints devised from an extended type ignore the extensibility and treat the type as if it were the normal non-extended declaration.

Proposed TTCN change 14: Any extensibility defined within an ASN.1 type is ignored within any associated constraint definitions and interpretation.

6.7 Automatic Tagging in TTCN

ASN.1:1997 introduces the feature of AUTOMATIC tagging. This provides a new tagging mode in addition to the existing IMPLICIT and EXPLICIT. When AUTOMATIC tagging is selected the system will automatically insert any necessary tags within the associated module without the need for user intervention (N.B. the user still has the choice to override the AUTOMATIC mechanism for specific constructs by explicitly defining tags).

AUTOMATIC tagging is selected from the ASN.1 module header, but current TTCN only allows ASN.1 type definitions not module definitions; as a consequence there is no way to select the tagging mode within TTCN, which by default is EXPLICIT.

If there is no support in TTCN for AUTOMATIC tagging, all ASN.1 type declarations taken from modules having the tagging mode set to AUTOMATIC, will have to be carefully rewritten into TTCN type definitions using EXPLICIT tagging, thus reproducing manually the AUTOMATIC tagging productions.

For complex and large types, this task is non-trivial, error-prone and time consuming.

The proposed solution is to introduce tagging regimes into TTCN ASN.1 type proforma, in order for the user to be able to set the tagging mode to be used in the table.

Proposed TTCN change 15: Extend TTCN language to include the ASN1_Tagging option i.e.:

```
ASN1_Type    ::=   Type [ASN1_Tagging]
ASN1_Tagging ::=   EXPLICIT TAGS  |
                   IMPLICIT TAGS  |
                   AUTOMATIC TAGS |
                   empty
```

If no tag type is defined, EXPLICIT tagging will be assumed.

Regarding the introduction of tagging in TTCN ASN.1 type proformas, it is necessary to insert a new entry in the headers to allow the specification of the tagging mode. The example below shows a possible presentation:

PDU Type Definition		
Type Name : REL PCO Type : Tag Mode : AUTOMATIC TAGS Comments : No need to insert tag types before the field types below		
Field Name	**Field Type**	**Comments**
PD	OCTET STRING[1]	Protocol discriminator
CR	CREF_TYPE	Call reference
UI	OCTET STRING[2..131]	User information

Figure 26: Automatic tagging in TTCN

6.8 Exception Identifiers in TTCN

It could be of benefit to be able to reference exception cases in ASN.1 type definitions when used in TTCN. Those exceptions would be for handling run-time errors and thus might be useful for feedback to TTCN tools e.g. through record in the log file.

The proposed approach is a simplest one that does not impact on TTCN language: Exception identifiers can be defined in TTCN ASN.1 type definitions (as allowed in X.680 [1]) but they are translated to comments in the TTCN.

6.9 List of proposed changes to TTCN

No.	Description
1	Field identifiers are mandatory in all ASN.1 constraints proformas.
2	TTCN Test Suite Constants must be passed into ASN.1 constraints definitions as actual parameters.
3	The use of expressions with ASN.1 constraints definitions shall be forbidden.
4	Only the value notation syntax defined within the X.680 [1] standard shall be used within ASN.1 constraints proformas. The redefinition of the ASN.1 production for DefinedValue in the TTCN BNF is deleted.
5	Constraints for ASN.1 types can be defined using TTCN tabular constraints
6	Extend TTCN tabular constraint proforma field name syntax to allow referencing to ASN.1 record structures using the existing RecordRef production.
7	Within TTCN tabular type definitions all fields are mandatory unless explicitly defined as optional.
8	Include the new string types BMPString, UniversalString and UTF8String in TTCN.
9	Add new proforma and BNF productions in TTCN to allow definition of information objects
10	Add new proformas and BNF productions to allow declaration of information objects and information object sets
11	The TTCN syntax for the parameter lists for all relevant proformas will conform to ASN.1:1997
12	Extend TTCN proformas and productions to include ASN.1 98 parameterization
13	Macro expansion in type definitions cannot be performed on parameterized types.
14	Any extensibility defined within an ASN.1 type is ignored within any associated constraint definitions and interpretation.
15	Extend TTCN language to include the ASN1_Encoding option

Table 2: List of changes to TTCN

7 Changes to ASN.1

7.1 Summary of ASN.1 issues

The following list summarises the issues for harmonising ASN.1-SDL/TTCN:

- ASN.1 identifiers;
- ASN.1 keywords.

7.2 Addition of 'underscore' to ASN.1 identifiers

Currently, it is possible to use hyphens in ASN.1 identifiers. This is defined in X.680 [1]:

> *an identifier name "shall consist of an arbitrary number (one or more) of letters, digits, and hyphens. The initial character shall be a lower-case letter. A hyphen shall not be the last character. A hyphen shall not be immediately followed by another hyphen".*

Because the hyphen sign is used in SDL as an operator, it is ambiguous when using ASN.1 (that does not include operators for type values) in combination with SDL. This means that ASN.1 identifiers cannot be used in SDL without removing or changing hyphens each time they appear. This is not desirable for the sake of harmonisation and integration between the languages.

Proposed ASN.1 change 1: Extend the definition of identifier name to allow the use of underscores as well as hyphens

7.2.1 Consequences and migration

The consequence of the proposed change is that ASN.1 modules conforming to the X.680 series specifications can be written in (or transformed too) a syntax which can be directly used within other high level specification languages. The proposal also retains full backwards compatibility with the existing ASN.1 language specification.

7.3 Alternative syntax for ASN.1 keywords

ASN.1 currently defines the following base types with space separators between the two keywords.

- OCTET STRING
- BIT STRING
- CHARACTER STRING
- OBJECT IDENTIFIER
- EMBEDDED PDV

Whilst doing this slightly reduces the number of keywords required within the language, it is not clear what further benefit it provides. when such base types are referenced in SDL, the type has to be translated to conform to the SDL syntax (the space is replaced by an underscore, for example). This is how it is defined within Z.105 [6]. The rationale is that otherwise these ASN.1 base types cannot be used in SDL without changes to remove or change the space each time they appear. This is not desirable for the sake of harmonisation and integration between the languages.

Proposed ASN.1 change 2: Extend the ASN.1 syntax to include an alternative set of keywords containing no space characters

7.3.1 Consequences and migration

The consequence of the proposed change is that ASN.1 modules conforming to the X.680 series specifications can be written in (or transformed too) a syntax which can be directly used within other high level specification languages. The proposal also retains full backwards compatibility with the existing ASN.1 language specification.

Migration would require replacing each occurrence of the base types when they appear in existing ASN.1 specifications. The base type will be changed so that the space is replaced by a hyphen (or underscore if the relevant proposal is accepted) to make the base types consist of a single keyword.

7.4 List of proposed changes to ASN.1

The following table lists the required changes to ASN.1(X.680 [1])

No.	Description
1	Extend the definition of identifier name to allow the use of underscores as well as hyphens
2	Extend the ASN.1 syntax to include an alternative set of keywords containing no space characters

Table 3: List of changes to ASN.1

Annex A: Proposals submitted to the ITU-T

A.1 SDL Case Sensitivity

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to Study Group 10 meeting

Berlin, July 2-3 1998

SOURCE: ETSI STF121

QUESTION: 6/10

TITLE: Make SDL case sensitive

Proposal

Extend the SDL language to make it case sensitive.

Rationale

SDL is currently defined as being case-insensitive as follows (Z.100, p16):

"In all <lexical unit>s except <character string>, <letter>s are always treated as if uppercase. (The treatment of <national>s may be defined by national standardisation bodies.)"

This causes many problems when using SDL with ASN.1, TTCN, C, C++, Java and other programming languages.

Examples of such problems include:

- using or referencing ASN.1 within SDL
- code generation from SDL, where the case sensitivity of identifiers, types, etc. are relevant

Proposed Z.100 change 1: Change Z.100 to make SDL case sensitive. It is further proposed to define a definite case for keywords (either upper- or lower case).

Consequences

This would require defining case for all keywords, since it would not be of benefit to define parts of the language as case insensitive. Since the current definition defines <letter>s to be treated in uppercase, it is proposed all SDL keywords are defined as uppercase. This also has the advantage of distinguishing keywords easily within SDL specifications. This would change the definition to:

"In all <lexical unit>s <letter>s are case sensitive. <keyword>s are defined in upper case."

All keywords currently defined in Z.100 would be defined in upper case, instead of lower case as currently written in Z.100.

Migration

Because case can be mixed in case-insensitive SDL throughout the specifications, identifiers and names must be defined consistently in case-sensitive SDL. Migration from case insensitive to case sensitive can be done through a series of rules. The basic requirements for migration of existing systems are:

- translate all <keyword>s into upper case
- for <character string>s and <note>s, no translation
- where ASN.1 is referenced, case shall be determined by the case of the ASN.1 definition, and this case used in all references and subsequent definitions
- for all other <word>s within SDL, case shall be determined by the case of definition, and this case used in all references and subsequent definitions

A.2 In-line ASN.1 in SDL

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to Study Group 10 meeting

Berlin, July 2-3 1998

SOURCE: ETSI STF121

QUESTION: 6/10

TITLE: Removal of use of in-line ASN.1 in SDL

Proposal

Removal of ability to define in-line ASN.1 types and values in the SDL language.

Rationale

Use of in-line ASN.1 in SDL causes many problems (some of which are identified in Z.105).

Removing in-line ASN.1 from SDL, will provide the following benefits:

- provide a cleaner interface between SDL and ASN.1 in accordance with the proposed paradigm
- many syntactical restrictions will be removed

Proposed Z.105 change 1: Remove in-line ASN.1 from Z.105 (and thus from SDL).

Consequences

- The removal of in-line ASN.1 will require the implementation of The CHOICE, OPTIONAL and DEFAULT constructs in SDL. Phase two of this contribution will address this issue, but in the meantime any proposals for how this may be done (e.g., syntax) would be appreciated.
- No ASN.1 definitions can explicitly appear in-line in the 'SDL part' of a specification. However, references to these ASN.1 types from the SDL is still possible. Migration and backward compatibility are defined as part of this proposal.
- Z.105 defines the need for a ';' at the end of in-line ASN.1 definitions. This is not required if either proposals for case-sensitivity in SDL or removal of in-line ASN.1 in SDL are accepted.
- The ASN.1 extensibility syntax (X.680 Amendment 1) will automatically be supported by this proposal in the sense that extensibility is an integral part of the ASN.1 notation.

Migration

Migration would require removing in-line ASN.1 from SDL specifications, and placing them in a separate ASN.1 module. The ASN.1 module corresponds to the SDL package i.e. it will contain type and value definitions. Note the following cases:

- Where ASN.1 types are defined in isolation (no mix of SDL and ASN.1) these can be defined in an ASN.1 module and referenced.

```
/* Definition of ASN.1 types in SDL text symbol */

Domain ::= ENUMERATED {
    hardware(0), software(1) };
```

→

```
-- Definition of ASN.1 module

ExhibitionModule DEFINITIONS ::=

Domain ::= ENUMERATED {
    hardware(0), software(1) }
```

↓

then using the IMPORTS keyword in SDL:

```
IMPORTS Domain, CompanyId
FROM ExhibitionModule;

or
```

Where SDL sorts are used in ASN.1 type definitions, these will have to be translated into pure SDL type definitions, with the relevant ASN.1 constructs defined as new type definitions in an ASN.1 module and referenced:

```
/* SDL excerpt */
NEWTYPE PhoneFax
    SEQUENCE OF INTEGER
ENDNEWTYPE PhoneFax;

/* and now the following inline ASN.1 type */
```

→

```
-- ASN.1 module
ExhibitionModule DEFINITIONS ::=

PhoneFax ::= SEQUENCE OF INTEGER

CompanyId ::= SEQUENCE {
```

↓

then using the IMPORTS keyword in SDL:

```
IMPORTS PhoneFax, CompanyId
FROM ExhibitionModule;
```

- Where ASN.1 is used in SDL types, these can be defined in an ASN.1 module and referenced:

In ASN.1, if we have:

```
ListOfDepts ::= SEQUENCE SIZE(2)  OF INTEGER
```

that is used in SDL as:

```
NEWTYPE CompanyCity
        nn ListOfDepts,
        city IA5String
```

becomes:

In ASN.1:

```
ListOfDepts ::= SEQUENCE SIZE(2)  OF INTEGER


CompanyCity ::= SEQUENCE {
```

that is referenced in the SDL:

```
IMPORTS ListOfDepts, CompanyCity FROM Asn1Module;
```

A.3 Adding Operators to ASN.1 Types

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to Study Group 10 meeting

Heidelberg, December 8-9 1998

SOURCE: ETSI STF121

QUESTION: 6/10

TITLE: Adding Operators to ASN.1 Types

Proposal

Extend reference clause (Z.105 import clause) to allow association of SDL operators with ASN.1 definitions

Rationale

To be able to use the ASN.1 types in an SDL environment it is essential there is a possibility to add operators to them without defining a new type. This is not a question of inheritance, but rather taking the approach that the type definition in ASN.1 is incomplete with respect to the behavioural aspects of the abstract data type.

The support in Z.105 today is not sufficient as the redefinition of the type (e.g. by inheritance) will create a new type which is not shared with other possible users of the same ASN.1 module.

Changes in Z.100 / Z.105

The solution proposed is to extend (associate) operators with ASN.1 types when they are imported / used. This will require changes to Z.100 and Z.105 to support both methods of reference

```
<package reference clause> ::=
        use <package name> [ / <definition selection list> ] <end> |
        use <package name> <type redefinition>* enduse <package name>

<definition selection list> ::=
        <definition selection> {, <definition selection>}*

<definition selection> ::=
        [<entity kind>] <name> [ <type redefinition> ]

<entity kind> ::=
        system type
    |   block type
    |   process type
    |   service type
    |   signal
    |   procedure
    |   newtype
    |   signallist
    |   generator
    |   synonym
    |   remote

<type redefinition> ::= type <type name> adding operators <operator definitions>
        endtype [<type name>]
```

Inheritance

It should be possible to inherit **import**ed / **use**d ASN.1 data types in the same manner as SDL sorts, and be able to add literals and operators to the new type definition. This would ensure that the only difference in usage between SDL sorts and ASN.1

type definitions is in the **use** clause, and the rationale for this difference has already been stated. Any operators associated with an ASN.1 definition when first referenced would also be available in any inherited types.

Limitations

For each distinct ASN.1 data type there must be at most one use construct to add operators to that type within an SDL system. This restriction is required to avoid operator inconsistency.

Examples

This is an example of how operators could be added to imported ASN.1-types

First the ASN.1 Module from which the definitions are imported.

```
MODULE MyModule ::=
BEGIN

Expert ::= SEQUENCE
    {
      name  NameType,
      hotel     HotelType,
      car       RentalCarType,
      hunger    INTEGER OPTIONAL,
      task  TaskType    OPTIONAL
    }
STF ::= SEQUENCE OF Expert

stf121 STF := { { Linus, Ambasador, NULL },
    { Ian, Ambasador, VWPolo } }
END MyModule;
```

Then the SDL Use Clause will be used to import the declarations into the SDL system using the transformation rules from Z.105. There are two ways to import an ASN.1 type and define operators for it depending on whether the use statement globally imports all types from a module or explicitly imports the type we wish to add operators to.

First consider the case when the use construct imports all types from a module. In this case a type redefinition is allowed to add operators.

```
Use myModule
Type Expert
adding
    operators
      assign:   Expert, Work -> Expert
      feed: Expert, Food -> Expert
      ready:    Expert -> Boolean
EndType Expert;

EndUse MyModule;

In the second alternative where you specify each type you want to use respectively you can add the
operators at the same time.

Use myModule/Expert
adding
    operators
      assign:   Expert, Work -> Expert
      feed: Expert, Food -> Expert
      ready:    Expert -Boolean
EndType Expert;

Use myModule/STF;

Use myModule/stf121;
```

Finally, an example of referencing the types and the operators.

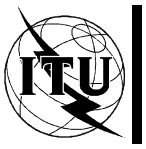
```
DCL     Linus, HappyLinus, MrHyde Expert;
DCL  destination HotelType;

TASK HappyLinus := feed(Linus, Pizza);

TASK    MrHyde := feed (stf121(1), Beer );

TASK destination := stf121(1)!hotel;
```

A.4 Adding Encoding Rule Reference to SDL



INTERNATIONAL TELECOMMUNICATION UNION

Contribution to Study Group 10 meeting

Heidelberg, December 8-9 1998

SOURCE: ETSI STF121

QUESTION: 6/10

TITLE: Adding Encoding Rule Reference to SDL

Proposal

Extend SDL to allow optional encoding rule reference to be specified for a signal parameter.

Rationale

SDL/Z.105 does not allow the definition of encoding rules. A protocol specified in ASN.1 for which encoding rules are given, e.g. BER, needs to have a corresponding mapping in SDL if a code generator is to work directly from SDL source alone.

Changes in Z.100

The proposal is split into two parts. The first part allows the definition of the encoding rules available. The second part allows reference to the required encoding rule for a particular signal parameter.

Defining the encoding rules available (optional)

This provides a way of defining all present encoding rules. This is for syntax checking in the referencing, preventing nasty spelling errors in the code. To do this we introduce a new construction in SDL.

```
<encoding>      ::= ENCODING ( <encoding rules list> )
<encoding rules list>   ::=     <encoding rule name> [ ',' <encoding rule  name> ]*
```

Example:

```
ENCODING (BER, PER, MyEncoding);
```

Now the encoding rules defined for the system can be referenced in the signal definitions. This way we can associate each of the signal definitions with the appropriate encoding rules.

Referencing encoding rules

The encoding rules must be specified for each signal parameter individually. The rationale for this is twofold.

1) There could be more than one encoding rule defined for a protocol.

2) In a stack oriented system only signal parameters representing PDU data sent to the environment should be encoded.

The grammar for the signal parameters is changed to:

```
<signal definition item> ::=
        <signal name>
        [<formal context parameters>]
        [<specialization>]
        [<signal parameter list>][<signal refinement>]
```

```
<signal parameter list> ::=
        (<signal parameter> {, <signal parameter>}*)
<signal parameter>::=   <sort> [ ( ENCODED BY <encoding rule name> ) ]
```

An Example of using the encoding rules for a signal parameter is.

```
SIGNAL NDataReq(
  NAddrType,
  NAddrType,
  NPDUData   (ENCODED BY BER ), /* Parameter Encoding */
  prioType);
```

A.5 Adding Z.105 String Value notation to Z.100

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to Study Group 10 meeting

Erlangen, December 8-9 1998

SOURCE: ETSI STF121

QUESTION: 6/10

TITLE: Adding Z.105 String Value notation to Z.100

Proposal

Extend the <character string literal> production to allow *"string"*, *' 'B* and *' 'H* value notation within Z.100.

Rationale

To be able to use the ASN.1 String types in an SDL environment it is essential there is succinct and easy way to define string values.

Changes in Z.100

String Types

To make the languages coherent, there is a need for new string types in SDL. These are the new string types defined in ASN.1 - 97

BMPString

UTF8String

Type Definition

The new strings should be added to the list of predefined types in the list in paragraph 5.1.1 in Z.100. This would give them the same status as the other predefined data types.

Value Notation

The <character string literal> production should be extended with the new string types.

The <character string> production should be kept as it is the base for the comments and informal text.

```
<character string literal> ::=  <character string> | <character string list> | <quadruple> |
                                <tuple> | <bitstring> | <hexstring> | <quoted string>

<character string list> ::= { <charsyms> }
<charsyms>  ::= <charsdfn> [, <chardfn> ]*
<chardfns>  ::= <cstring> |<quadruple> | <tuple> | <defined value>

<quadruple> ::={ <group> , <plane> , <row> , <cell> }
<group> ::= <number>
<plane> ::= <number>
<row> ::= <number>
<cell> ::= <number>

<tuple> ::= { <table column>, <table row> }
<table column> ::= <number>
<table row>         ::= <number>
```

```
<quoted string> ::=     " <text> "
<bitstring> ::=  <apostrophe> { 0 |  1  }* <apostrophe> B
<hexstring> ::=  <apostrophe> {<decimal digit> |  A  |  B  |  C  |  D  |  E  |  F }*
                 <apostrophe>  H
```

For example:

```
DCL special BMPString   := “A char = “// {1,2,3,4} // “.”;
DCL address OCTETSTRING := ‘FFFF’H;
DCL address BITSTRING   := ‘1101’B;
```

A.6 Addition of ‘underscore’ to ASN.1 identifiers

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to joint ISO / ITU ASN.1 meeting

Lannion, January 18-28 1999

SOURCE: ETSI STF121

TITLE: Addition of ‘underscore’ to ASN.1 identifiers

Proposal

Extend the definition of identifier name to allow the use of underscores as well as hyphens.

Rationale

Currently, it is only possible to use hyphens as word separators in ASN.1 identifiers.

Because the hyphen sign is used in many high level languages as an operator, this causes ambiguity and parsing problems when using ASN.1 in combination with these languages. For example In the case of SDL this means that ASN.1 identifiers cannot be used in SDL without removing or changing hyphens each time they appear.

Clearly it is desirable that the ASN.1 language allows direct integration with other specification languages used to define the associated semantics of a syntax. The extension of the ASN.1 identifiers to include underscores will allow this direct integration.

Changes in X.680

Change the definition for identifier name to:

an identifier name “shall consist of an arbitrary number (one or more) of letters, digits, hyphens and underscores. The initial character shall be a lower-case letter. A hyphen or underscore shall not be the last character. A hyphen shall not be immediately followed by another hyphen. An underscore shall not immediately followed by another underscore. Underscores and hyphens may not be mixed within a single identifier name”.

A.7 Alternative syntax for ASN.1 keywords

INTERNATIONAL TELECOMMUNICATION UNION

Contribution to joint ISO / ITU ASN.1 meeting

Lannion, January 18-28 1999

SOURCE: ETSI STF121

TITLE: Alternative syntax for ASN.1 keywords

Proposal

Extend the ASN.1 syntax to include an alternative set of keywords containing no space characters.

Rationale

The ASN.1 language specification currently defines a set of base types with space separators between the two keywords. This space separator causes problems when trying to integrate ASN.1 definitions into other high level specification languages.

When such base types are referenced in SDL, the type has to be translated to conform to the SDL syntax (the space is replaced by an underscore).

The extension of the ASN.1 syntax to allow the an alternative set of keywords not containing space separators will allow direct integration of ASN.1 definitions into other high level specification languages.

Changes in X.680

Add the following set of keywords:

OCTET_STRING

BIT_STRING

CHARACTER_STRING

OBJET_IDENTIFIER

EMBEDDED_PDV

where

OCTET_STRING ≡ OCTET STRING

BIT_STRING ≡ BIT STRING

CHARACTER_STRING ≡ CHARACTER STRING

OBJECT_IDENTIFIER ≡ OBJECT IDENTIFIER

EMBEDDED_PDV ≡ EMBEDDED PDV

A.8 Proposal Status

Proposal	Standards Body	Submitted	Status
SDL Case Sensitivity	ITU-T, SG10, Q.6	July 98	Proposal accepted for SDL 2000, with the refinement that SDL keywords must not be written in mixed case (i.e. all in upper case or all in lower case)
In-line ASN.1 in SDL	ITU-T, SG10, Q.6	July 98	Concept accepted with the proviso that a standard including in-line ASN.1 type declaration is still required in the short term. The proposed solution is to have two parallel standards Z.105 will be the new developed grammar without in-line ASN.1 and Z.107 will still allow in-line ASN.1 type definitions.
Operators for ASN.1 Types	ITU-T, SG10, Q.6	Dec 98	The proposal was rejected in the proposed syntax. However support for defining operators for ASN.1 types within SDL without generating a new type was added to the requirements of the new SDL 2000 data model.
SDL Encoding Rules	ITU-T, SG10, Q.6	Dec 98	The need for encoding reference was accepted by the experts group, but the proposed syntax and scope were rejected. Encoding specification was added to the SDL open items list to be resolved at a later date.
Extend Z.100 String Notation	ITU-T, SG10, Q.6	Dec 98	This proposal was accepted and will be integrated into the new Z.100 for SDL 2000. The octal and binary value notation will also be added to integer types.
Addition of ‘underscore’ to ASN.1 identifiers	ISO/ITU-T	Jan 99	
Alternative syntax for ASN.1 keywords	ISO/ITU-T	Jan 99	

History

Document history		
0.1	June 1998	First draft
0.2	September 1998	Second draft presented to MTS
0.3	December 1998	Third revision (after MTS meeting October 1998)
0.4	January 1999	Fourth revision incorporating comments on 0.3 document plus minor editorials. Input to MTS #28 for approval.
0.5	March 1999	Final draft after addressing comments from MTS#28 23-25 March 1999 on version 0.0.4. Placed on ETSI server March 29 1999 for approval by correspondence by end of April 1999.