Chapter

14

The ASN.1 Utilities

The ASN.1 Utilities perform three main functions in Telelogic Tau:

- **They can translate an ASN.1 module to an SDL package. This makes it possible to use ASN.1 types and values in SDL.**
- **They make it possible for the TTCN suite to retrieve external ASN.1 types and values that are used in TTCN.**
- **They produce type information for BER coders in SDL.**

Note: ASN.1 support in the TTCN suite

The TTCN to C compiler supports only a limited subset of ASN.1. See “TTCN ASN.1 BER Encoding/Decoding” on page 56 in chapter 2, *Release Notes, in the Release Guide* for further details on the restrictions that apply.

This chapter is the reference manual for the ASN.1 Utilities.

Introduction

This chapter describes the ASN.1 Utilities. It is assumed that the reader is familiar with ASN.1.

Application Areas for the ASN.1 Utilities

The main foreseen applications of the ASN.1 Utilities are the following:

- A lot of telecommunication protocols and services are defined using ASN.1. The ASN.1 Utilities make it easier to specify and implement these with SDL.
- The ASN.1 Utilities enable the SDL suite and the TTCN suite to share common data types by specifying these in a separate ASN.1 module.
- The ASN.1 Utilities generate type information for BER encoding/decoding for the SDL suite.

Overview of the ASN.1 Utilities

The ASN.1 Utilities support the following main functions:

1. Perform syntactic and semantic analysis of ASN.1 modules.
2. Generate SDL code from ASN.1 modules.
3. Extract the ASN.1 types and values which are referred in the TTCN suite.
4. Generate type information for BER encoding and decoding for the SDL suite.

For further information about BER encoding and decoding, see chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual*.

In normal cases, the ASN.1 Utilities are completely hidden for the user by the SDL Analyzer and the TTCN Analyzer.

From the user's point of view, an ASN.1 module is very similar to an SDL package: ASN.1 data types can be defined in a module, and then be used within SDL, using operators that are defined in ITU Recommendation Z.105. When an SDL system containing ASN.1 modules is analyzed, the Analyzer will order the ASN.1 Utilities to translate these modules into corresponding SDL packages.

In the TTCN suite, indirect use of the ASN.1 Utilities is made by the ASN.1-by-reference table. When such a table is analyzed, the TTCN suite orders the ASN.1 Utilities to extract the ASN.1 types and values in a specified ASN.1 module. For more information about this functionality, see “ASN.1 External Type/Value References” on page 1188 in chapter 27, *Analyzing TTCN Documents (on UNIX)*.

Using the ASN.1 Utilities

The ASN.1 Utilities are implemented in the executable `asn1util`. `asn1util` can be used in two ways:

1. Stand-alone from the organizer (command-line interface).
2. Via the PostMaster

Command-Line Interface

Usage: `asn1util [options] { <file> [options] }*`

Option	**Meaning**
`-h`	display a help message
`-v`	display version
`-q`	be quiet, suppress some output messages
`-c`	generate encode/decode type information for the SDL Suite and asn1_cfg.h configuration file
`-g`	generate coder information for TTCN suite
`-B`	set BER as default encoding
`-P`	set PER as default encoding
`-N <name>`	set <name> as default encoding
`-m`	include module name in encode/decode type nodes and macros

Option	Meaning
`-n <name>`	use <name> for the name of the interface (*.ifc) files
`-S <config>`	use <config> for type names configuration in (*.ifc) files
`-s <file>`	generate SDL output in `<file>`
`-a`	append the output to an existing file instead of creating a new file
`-b`	generate SDL body only, i.e. do not generate package headings (makes it possible to import generated SDL with #INCLUDE)
`-r`	generate references (#SDTREF) to source file
`-e`	generate all operators for the SDL enumerated type as listed in Z.105. Default is to emit some of the operators in Z.105
`-O`	generate values for SDL Make operator with optional and default support
`-u <package>`	add “use <package>;“ to all generated SDL packages
`-J <name> <files>`	Join all ASN.1 modules from <files> into one SDL package <name> (see Example 29 on page 699)
`-K <file>`	Perform substitution for keywords listed in <file> (see “Keywords substitution” on page 701)
`-i <file>`	generate TTCN output to <file>
`-l <file>`	take command line from <file>
`-post`	wait for commands via the PostMaster (see “PostMaster Interface” on page 700)
`-T<dir>`	put generated code in directory `<dir>`

Example 27 ––

```
asn1util -r -s myfile.pr -c myfile.asn
(myfile.asn contains ASN.1 module MyModule)
```

–––

The command in the example translates the module MyModule in file myfile.asn to an SDL package MyModule in file 'myfile.pr'. The generated package will contain backward references to the source file 'myfile.asn'. Encode/decode type nodes are generated in C-source file 'MyModule_asn1coder.c' and C-header file 'MyModule_asn1coder.h'. A configuration file "asn1_cfg."' with compile switches for coder related files is generated.

Example 28 ––

```
asn1util AsnModule1.asn AsnModule2.asn
```

–––

If no options are specified, then asn1util only performs syntactic and global semantic analysis for AsnModule1.asn and AsnModule2.asn, no output is generated.

If no input file is specified, then asn1util does nothing except showing help or version number if correspondent options are specified.

Example 29 Joining modules –––––––––––––––––––––––––––––––––

```
asn1util -J Join-Module -s my.pr my1.asn my2.asn
```

–––

The ASN.1 modules from the files my1.asn and my2.asn will be joined together in the SDL package Join_Module. Name clashes may occur if the same name is available in different ASN.1 modules the filed are joined. These problems are resolved according to a set of name clash resolving rules, see <u>"Joining modules" on page 704</u>.

Configuration file generation

For the `-c` option encode and decode type information is generated to C-files. Also asn1util performs the analysis of ASN.1 types and some features used in the specification and generates file `asn1_cfg.h`.

This file contains compile switches that are referenced from inside the coders code. When `asn1_cfg.h` is used by the build process the preprocessor automatically throws away useless parts of the code from encoding and decoding related files. This helps to reduce the code size and improve the performance of encoding and decoding procedures.

For example, if the ASN.1 file does not contain OCTET STRING and SET OF types in the module, the following definitions

```
#ifdef CODER_AUTOMATIC_CONFIG
#define CODER_NOUSE_OCTET_STRING
#define CODER_NOUSE_SET_OF
#endif
```

will be included in the configuration file.

This feature can be turned on by the `CODER_AUTOMATIC_CONFIG` compile switch. For more information about available compile switches for the configuration see “Encoding configuration” on page 2820 in chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite*.

PostMaster Interface

The ASN.1 Utilities can also be invoked via the PostMaster. An example of this is when an SDL system that uses ASN.1 modules is analyzed. The Analyzer will then order the ASN.1 Utilities, via the PostMaster, to perform a translation of the ASN.1 modules to SDL packages. For a complete description of the PostMaster, see chapter 11, *The PostMaster*.

On UNIX, the PostMaster communication may also be invoked by starting `asn1util` with the `-post` command-line option. `asn1util` will then wait for commands sent to it from the PostMaster.

Translation of ASN.1 to SDL

This section describes the detailed translation rules from ASN.1 to SDL that are implemented in the ASN.1 Utilities. The translation rules all conform to Z.105, except for the cases described in “Restrictions to Z.105” on page 31 in chapter 2, *Release Notes, in the Release Guide*.

General

- Case sensitivity is according to Z.105, i.e. ASN.1 names are converted directly to SDL names. This implies that in rare cases, correct ASN.1 modules may cause name conflicts when used in SDL.

Note:

Since named numbers, named bits, and integer values are all mapped to integer synonyms, the same name should not be used more than once, because this will lead to name conflicts in SDL.

- ‘-’ (dash) in ASN.1 names is transformed to ‘_’ (underscore), e.g. `long-name` in ASN.1 is transformed to `long_name` in SDL.
- In accordance with Z.105, tag information is ignored in the translation to SDL.
- As SDL does not have “in-line types”, one ASN.1 type may be mapped to more than one SDL type. The generated in-line types get dummy names.
- External type/value references are mapped to qualifiers. For example `A.a` is mapped to `<<package A>> a`. Also a use clause (`use A;`) is generated.

Keywords substitution

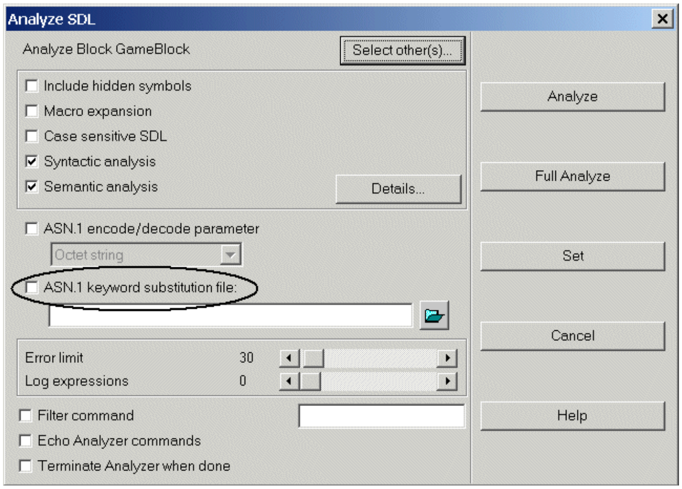
ASN.1 generators can be configured to be sensitive to a certain number of identifiers. There is a special text file named ‘asn1util_kwd.txt’ that contains a list of identifiers and a list of their substitution during mapping. By default this file is used to configure target languages keywords substitution. It can be edited to get another functionality or another set of keywords to be replaced.

‘asn1util_kwd.txt’ should contain pairs of identifiers where the first one is the identifier from original ASN.1 specification that will be replaced by the second identifier during generation. ‘asn1util_kwd.txt’ should conform to the following syntax:

Example 30 Configuration file syntax

```
<identifier1>   <identifier1 substitution>
<identifier2>   <identifier2 substitution>
             ...
<identifierN>   <identifierN substitution>
```

ASN.1 Utility reads the first configuration file it finds. It searches for the ‘asn1util_kwd.txt’ file first in the current folder, then in the home folder and finally in the installation. If a configuration file named ‘asn1util_kwd.txt’ is put in the home folder or in the current working folder, it will override the default configuration file from the installation. The configuration file to be used can also be specified in the Analyze dialog or from the command line with the '-K' option (see "Command-Line Interface" on page 697).


Analyze SDL

Analyze Block GameBlock

- [ ] Include hidden symbols
- [ ] Macro expansion
- [ ] Case sensitive SDL
- [x] Syntactic analysis
- [x] Semantic analysis
- [ ] ASN.1 encode/decode parameter: Octet string
- [ ] ASN.1 keyword substitution file:

Error limit 30

Log expressions 0

- [ ] Filter command
- [ ] Echo Analyzer commands
- [ ] Terminate Analyzer when done

Figure 168: Keywords substitution

'asn1util_kwd.txt' is always present in the installation and it is configured to replace keywords from SDL, TTCN, C and C++ languages: the ASN.1 identifier `name`, that is a keyword in SDL, TTCN or C++, is replaced by `name_SDL_KEYWORD`, `name_TTCN_KEYWORD` or `name_CPP_KEYWORD` to avoid syntax errors in the target languages (see "Appendix A: List of recognized keywords" on page 755). If an original ASN.1 identifier has been modified, a warning message is reported. See "ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2" on page 754).

Note: Keywords recognition

ASN.1 in case-sensitive language and target language keywords are also recognized in case-sensitive mode. If generated SDL is analyzed in case-insensitive mode, there could still be keyword problems left. For example, ASN.1 contains the type named `Start` and it will not be recognized to be an SDL keyword, because the keyword `start` will be compared, but in case-insensitive mode `Start` is still a keyword in SDL, which will result in syntax errors.

Example 31 Keywords default substitution –––––––––––––––––––––

For these ASN.1 definitions:

```
CASE ::= ENUMERATED { upper, lower }

T ::= SEQUENCE {
        int INTEGER,
        explicit BOOLEAN,
        case     CASE,
        signal   INTEGER
}

value1 T ::= {
        int 5,
        explicit TRUE,
        case lower,
        signal 27 }
```

With default keywords substitution file the following SDL is generated:

```
newtype CASE_TTCN_KEYWORD
  literals upper,lower
  operators
    ordering;
endnewtype;

newtype T struct
  int_CPP_KEYWORD  Integer;
  explicit_CPP_KEYWORD  Boolean;
  case_CPP_KEYWORD  CASE_TTCN_KEYWORD;
  signal_SDL_KEYWORD  Integer;
endnewtype;

synonym value1 T = (. 5, true, lower, 27 .);
```

–––

A configuration file allows the user to control the set of keywords to be replaced. Removing lines with TTCN keywords, for example, will switch off TTCN keywords sensitivity. Providing an empty configuration file will result in switching off keywords substitution completely.

Module

- An ASN.1 module is translated to an SDL package as specified in Z.105. The `DefinitiveIdentifier` (object identifier after module name) is ignored. The tag default is also ignored.
- `EXPORTS` is mapped to a corresponding `interface`-clause.
- `IMPORTS` is mapped to a corresponding package reference clause. The `AssignedIdentifier` (object identifier after module name) is ignored.

Note:

On UNIX, the `-b` option disables generation of package/endpackage, interface and use clauses. Files that have been generated this way can be included in SDL with the #INCLUDE directive, see *"Including PR Files" on page 2436 in chapter 55, The SDL Analyzer*.

Example 32 ––

```
MyModule DEFINITIONS ::= BEGIN

EXPORTS A, b, C;
IMPORTS X, Y, z FROM SomeModule { iso 3 0 8 }
...
END
```

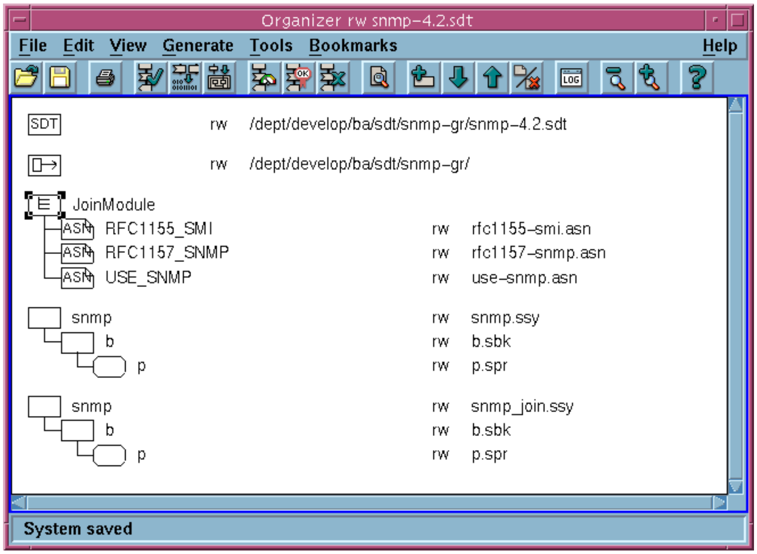
is mapped to

```
package MyModule;
interface newtype A, synonym b, newtype C;
use SomeModule / newtype X, newtype Y, synonym z;
...
endpackage;
```

––

Joining modules

Mapping for ASN.1 original module structure can be changed by applying joining module functionality. Several ASN.1 modules can be generated into one SDL package, if ASN.1 modules are arranged into groups in the Organizer. Joining modules can also be controlled from the command line (see "Command-Line Interface" on page 697).


Organizer rw snmp-4.2.sdt
File Edit View Generate Tools Bookmarks Help

```
SDT            rw   /dept/develop/ba/sdt/snmp-gr/snmp-4.2.sdt
               rw   /dept/develop/ba/sdt/snmp-gr/
JoinModule
  RFC1155_SMI       rw   rfc1155-smi.asn
  RFC1157_SNMP      rw   rfc1157-snmp.asn
  USE_SNMP          rw   use-snmp.asn
snmp                rw   snmp.ssy
  b                 rw   b.sbk
    p               rw   p.spr
snmp                rw   snmp_join.ssy
  b                 rw   b.sbk
    p               rw   p.spr
System saved
```

Figure 169: Joining modules

Joining means throwing away all import/export clauses and module headers and generating all module bodies into one big package with the name specified in the Organizer or command line interface.

When joining definitions from several ASN.1 modules into one SDL package, names in the resulting SDL package change according to the following rules:

- Clashed names are prefixed by the original ASN.1 module name
- Package names in the external references are replaced by the join package name

Example 33 ---

```
M1
DEFINITIONS ::=
BEGIN

IMPORTS S2 FROM M2;

T ::= SET OF SEQUENCE { a S2 }
S1 ::= IA5String
```

```
END

M2
DEFINITIONS ::=
BEGIN

T ::= SEQUENCE OF SEQUENCE OF M1.S1
S2 ::= BOOLEAN

END
```

without joining applied ASN.1 modules M1 and M2 are mapped to

```
use M2/
  newtype S2;
package M1; /*#ASN.1 'M1'*/

newtype T
   Bag(T_INLINE_0)
endnewtype;

newtype T_INLINE_0 /*#SYNT*/ struct
  a  S2;
endnewtype;

syntype S1 = IA5String endsyntype;

endpackage M1;

package M2; /*#ASN.1 'M2'*/

newtype T
   String (T_INLINE_0, emptystring)
endnewtype;

newtype T_INLINE_0 /*#SYNT*/
   String (<<package M1>>S1, emptystring)
endnewtype;

syntype S2 = Boolean endsyntype;

endpackage M2;
```

with joining to package Join-Package applied ASN.1 modules are mapped to

```
package Join_Package; /*#ASN.1 'Join_Package'*/

newtype M1_T
  Bag(M1_T_INLINE_0)
endnewtype;
newtype M1_T_INLINE_0 /*#SYNT*/ struct
```

```
     a  S2;
   endnewtype;
   syntype S1 = IA5String endsyntype;

   newtype M2_T
      String (M2_T_INLINE_0, emptystring)
   endnewtype;
   newtype M2_T_INLINE_0 /*#SYNT*/
      String (<<package Join_Package>>S1, emptystring)
   endnewtype;
   syntype S2 = Boolean endsyntype;

   endpackage Join_Package;
```

General Type and Value Assignment

A type assignment is mapped to a newtype or a syntype, depending on the type on the right-hand side of the '`::=`'. Tags are ignored. An ASN.1 value assignment is mapped to a synonym.

Example 34 ---

```
   T1 ::= INTEGER
   T2 ::= [APPLICATION 28] T1
   a BOOLEAN ::= TRUE
```

is mapped to

```
   syntype T1 = Integer endsyntype;
   syntype T2 = T1 endsyntype;
   synonym a Boolean = True;
```

Inline types naming

The ASN.1 language can use type definitions inside composite types, which are called inline types. Inline types are not allowed in SDL. In SDL, only named types can be used in a composite type. Implicit names are assigned to ASN.1 inline types and they are referenced by this name in SDL.

Implicit names for generated SDL have the following syntax: <parent_definition_name>_INLINE_<counter>, where parent_definition_name is either the name of the parent type or the name of the parent value, depending on if inline type exists in type or value assignment construct in ASN.1.

Example 35

```
T1 ::= SEQUENCE {
        a SET OF INTEGER,
        b CHOICE { x BIT STRING,
                   y OCTET STRING },
        c ENUMERATED { sat, sun } }

For type T the following inline types will be
generated to SDL

newtype T1 struct
  a  T1_INLINE_0;
  b  T1_INLINE_1;
  c  T1_INLINE_2;
endnewtype;

newtype T1_INLINE_0 /*#SYNT*/
   Bag(Integer)
endnewtype;

newtype T1_INLINE_1 /*#SYNT*/ choice
  x  Bit_string;
  y  Octet_string;
endnewtype;

newtype T1_INLINE_2 /*#SYNT*/
  literals sat,sun
  operators
    first: T1_INLINE_2 -> T1_INLINE_2;
    last:  T1_INLINE_2 -> T1_INLINE_2;
    succ:  T1_INLINE_2 -> T1_INLINE_2;
    pred:  T1_INLINE_2 -> T1_INLINE_2;
    num:   T1_INLINE_2 -> Integer;
    ordering;

  <operator definitions>

endnewtype;
```

Example 36

```
T2 SEQUENCE OF INTEGER ::= { {1,1} | {2,2} }

For T2 the following inline types will be generated
to SDL

newtype T2 /*#SYNT*/
   String (Integer, emptystring)
   constants ((. 1, 1 .)), ((. 2, 2 .))
endnewtype;
```

Example 37

```
val BIT STRING ( SIZE(3) ) ::= ‘101’B

synonym val val_INLINE_0 = bitstr(’101’);

syntype val_INLINE_0 = Bit_string constants size (3)
endsyntype;
```

Note:

SDL inline names can change if you change within the parent type or value in the ASN.1 specification, the counter can differ. If these names are used within an SDL system, then you must update the SDL system.

Boolean, NULL, and Real

`BOOLEAN, NULL` and REAL are mapped to the corresponding SDL types. Value notations for these types are mapped as follows

ASN.1 type	ASN.1 value	Corresponding SDL value
ANY		not supported (conform Z.105)
BOOLEAN	TRUE FALSE	True False
NULL	NULL	NULL
REAL	0 PLUS-INFINITY MINUS-INFINITY { mantissa 31416, base 10, exponent -4 }	0.0 PLUS_INFINITY MINUS_INFINITY 3.1416

If a REAL value has an exponent bigger than 1000 and if the mantissa is not zero, then it is mapped to PLUS_INFINITY or MINUS_INFINITY. If a REAL value has an exponent less than -1000, then it is mapped to 0.

Bit String

`BIT STRING` is mapped to the Z.105-specific type `Bit_string`. Named bits are mapped to integer synonyms. Values for bit strings are mapped to `hexstr`/`bitstr` expressions.

Example 38 ––

```
B ::= BIT STRING { bit0(0), bit23(23) }

b1 BIT STRING ::= ‘011 1110’B
b2 BIT STRING ::= ‘3AFC’H
```

is mapped to

```
syntype B = Bit_string endsyntype;
synonym bit0 Integer = 0;
synonym bit23 Integer = 23;

synonym b1 Bit_string = bitstr(‘0110 1110’);
synonym b2 Bit_string = hexstr(‘3AFC’);
```

––

Note:

`Bit_string`, as opposed to most other string types in SDL, has indices starting with 0!

Type `Bit` is a Z.105 specific type with literals 0 and 1, and with boolean operators.

Available operators:

```
bitstr    : Charstring             -> Bit_string;
     /* converts a Charstring consisting of ‘0’ and
        ‘1’-s to a Bit_string */
hexstr    : Charstring             -> bit_string;
     /* converts a Charstring consisting of
        hexadecimal characters to a bit_string */
“not”     : Bit_string             -> Bit_string;
“and”     : Bit_string, Bit_string -> Bit_string;
“or”      : Bit_string, Bit_string -> Bit_string;
“xor”     : Bit_string, Bit_string -> Bit_string;
“=>”      : Bit_string, Bit_string -> Bit_string;
     /* bitwise logical operators */
mkstring  : Bit                    -> Bit_string;
length    : Bit_string             -> Integer;
first     : Bit_string             -> Bit;
last      : Bit_string             -> Bit;
“//”      : Bit_string, Bit_string -> Bit_string;
extract   : Bit_string, Integer    -> Bit;
modify!   : Bit_string, Integer, Bit-> Bit_string;
substring : Bit_string, Integer, Integer ->
```

```
                                Bit_string;
/* normal String operators, except that index
   starts with 0;
   see also “Sequence of Types” on page 716 */
```

Character Strings

`PrintableString`, `NumericString`, `VisibleString`, and `IA5String` (i.e. all ASN.1 character string types with character sets that are a subset of ASCII) are mapped to syntypes of SDL `Charstring`. Values for these strings are mapped to corresponding `Charstring` synonyms in SDL.

The same operators as for `Charstring` are available for these types, and values of these types can be assigned freely to each other without need for conversion operators.

For example, in SDL an `IA5String` value can be assigned to a `NumericString` variable (given that the `IA5String` only contains numeric characters).

Choice Types

A `CHOICE` type is mapped to the choice-construct that is described in more detail in “Choice” on page 2600 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*.

Example 39 ––

```
C ::= CHOICE {
        a INTEGER,
        b BOOLEAN}

c C ::= a:7
```

is mapped to

```
newtype C choice
   a Integer;
   b Boolean;
endnewtype;

synonym c C = a:7
```

––

The operators that are available for a `CHOICE` type are (assuming that `C` is defined as in Example 39 above):

```
aextract!       : C -> Integer;
```

```
     /* e.g. c!a returns 7 */
bextract!       : C -> Boolean;
     /* but c!b gives dynamic error! */
amake!          : Integer -> C;
bmake!          : Boolean -> C;
     /* build choice value, e.g. in SDL
        it is possible to write b:True */
amodify!        : C, Integer -> C;
     /* e.g. var!a := -5 */
bmodify!        : C, Boolean -> C;
presentextract! : C -> xxx;
     /* returns the selected field.
        xxx is an anonymous type with values a and 
b.
        E.g. c!present gives a */
```

Enumerated Types

An `ENUMERATED` type is mapped to a newtype with a set of literals plus some operators. By default only ordering operators are generated, use command line option -e to get the rest. The list of literals that is generated is reordered in accordance with the associated integer values.

Example 40 –––

```
N ::= ENUMERATED { yellow(5), red(0), blue(6) }
```

is mapped to (only signature of operators shown)

```
newtype N
  literals red, yellow, blue
  /* note that the literals have been reordered! */
operators
   ordering;
   first: N -> N;
   last: N -> N;
   succ: N -> N;
   pred: N -> N;
   num: N -> Integer;
 endnewtype,
```

–––

The operators that are available for an `ENUMERATED` type are (assuming that `N` is defined as in Example 40 above):

```
num : N -> Integer;
     /* num(yellow)=5, num(red)=0, num(blue)=6 */
"<" : N, N -> Boolean;
"<=": N, N -> Boolean;
">" : N, N -> Boolean;
">=": N, N -> Boolean;
    /* comparison based on num, i.e. red < yellow */
```

```
pred: N -> N;
succ: N -> N;
     /* predecessor/successor based on num, i.e.
        succ(red)=yellow, succ(yellow)=blue,
        pred(red) gives a dynamic error */
first: N -> N;
last : N -> N;
     /* first/last element based on num, i.e.
        first(red)=red, last(red)=blue */
```

Integer

`INTEGER` is mapped to the SDL `Integer` type, and ASN.1 integer values are mapped to corresponding SDL values.

Named numbers are mapped to synonyms.

Example 41 ––

```
A ::= INTEGER { a(5), b(7) }
```

is mapped to

```
syntype A = Integer endsyntype;
synonym a Integer = 5;
synonym b Integer = 7;
```

–––

Object Identifier

`OBJECT IDENTIFIER` is mapped to the Z.105-specific type `Object_Identifier`. The normal String operators are available for `Object_Identifier`, listed also in “Sequence of Types” on page 716. Indices start as usual with 1.

Octet String

`OCTET STRING` is mapped to the Z.105-specific type `Octet_string`. `Octet_string` is based on type `Octet`. This type is further described in “SDL Predefined Types” on page 2588 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*. The mapping for the octet string value notation to SDL is identical to bit strings, see “Bit String” on page 710.

Note:

`Octet_string`, has indices starting with 1.

Operators available:

```
bitstr       : Charstring       -> Octet_string;
hexstr       : Charstring       -> Octet_string;
     /* conversion from Charstring to Octet_string,
        see also “Bit String” on page 710*/
bit_string   : Octet_string     -> Bit_string;
octet_string: Bit_string        -> Octet_string;
     /* conversion operators
        Octet_string <-> Bit_string */
mkstring     : Octet            -> Octet_string;
length       : Octet_string     -> Integer;
first        : Octet_string     -> Octet;
last         : Octet_string     -> Octet;
“//”         : Octet_string, Octet_string ->
                                          Octet_string;
extract!     : Octet_string, Integer -> Octet;
modify!      : Octet_string, Integer, Octet ->
                                          Octet_string;
substring    : Octet_string, Integer, Integer ->
                                          Octet_string;
     /* normal String operators, see also
        “Sequence of Types” on page 716 */
```

Sequence/Set Types

`SEQUENCE` and `SET` are both mapped to SDL *struct*. From an SDL point of view there is no difference between `SEQUENCE` and `SET`. In order to support optional and default components, SDL has been extended with corresponding concepts.

Note:

Optional and default fields in *struct* are both non-standardized extensions to SDL.

Values are mapped to the “(. ... .)” construct (= `Make!` operator). Values for optional and default components are not supported. Instead, SDL tasks should be used to assign optional and default components.

Example 42 –––

```
S ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN,
        c IA5String DEFAULT “xyz” }

s S ::= { b TRUE }
```

is mapped to

```
newtype S struct
   a Integer optional;
   b Boolean;
   c IA5String := ‘xyz’;
endnewtype;

synonym s S = (. True .);
```

–––

The operators that are available for a `SEQUENCE` or `SET` type are (assuming that `S` is defined as in Example 42 above):

```
make!    : Boolean -> S;
     /* builds a value for S */
aextract!: S -> Integer;
bextract!: S -> Boolean;
cextract!: S -> IA5String;
   /* Extract operators. Note that aextract! gives
      dynamic error if the field has not been set */
amodify! : S, Integer -> S;
bmodify! : S, Boolean -> S;
cmodify! : S, IA5String -> S;
     /* Modify operators change one component
        in a Sequence/Set */
apresent : S -> Boolean;
     /* gives True if component a has been assigned
        a value, e.g. aPresent(s) = False */
```

Sequence of Types

`SEQUENCE OF` is mapped to the `String` generator. Values are mapped to corresponding synonyms.

Example 43 –––––––––––––––––––––––––––––––––––––––

```
S ::= SEQUENCE OF INTEGER
s1 S ::= { 3, 2, 5 }
s2 S ::= {}
```

is mapped to

```
newtype S
   String (Integer, Emptystring)
endnewtype;

synonym s1 S = (. 3, 2, 5 .);
synonym s2 S = (. .);
```

–––

The normal String operators are available for Sequence types. Indices start at 1.

The operators that are available for a `SEQUENCE OF` type are (assuming that `S` is defined as in Example 43 above):

```
mkstring  : Integer             -> S;
     /* make a sequence of one item */
length    : S                   -> Integer;
     /* returns number of elements in sequence */
first     : S                   -> Integer;
     /* returns first element in sequence */
last      : S                   -> Integer;
     /* returns last element in sequence
"//"      : S, S                -> S;
     /* returns concatenation of two sequences */
extract!  : S, Integer          -> Integer;
     /* returns the indexed element */
modify!   : S, Integer, Integer -> S;
     /* modify the indexed element */
substring : S, Integer, Integer -> S;
     /* Substring(S, i, l) returns substring of S
        of length l, starting at index i */
make!     : * Integer           -> S;
      /* adds the included elements to the string,
       * corresponds to (. .) */
append    : in/out S, Integer;
      /* appends one element to the string */
```

Set of Types

SET OF is mapped to the Z.105 specific Bag generator. For a more complete description of the Bag generator, see *“Bag” on page 2603 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.

Example 44 –––

```
S ::= SET OF INTEGER
s1 S ::= { 2, 2, 5 }
s2 S ::= {}
```

is mapped to

```
newtype S
   Bag (Integer)
endnewtype;

synonym s1 S = (. 2, 2, 5 .);
synonym s2 S = (. .);
```

–––

The operators that are available for a SET OF type are (assuming that S is defined as in Example 44 above):

```
incl     : Integer, S   -> S;
     /* add an element to the bag */
del      : Integer, S   -> S;
     /* delete one element */
incl     : Integer, in/out S;
del      : Integer, in/out S;
length   : S             -> Integer;
     /* returns number of elements */
take     : S             -> Integer;
     /* return some element from the bag */
take     : S,   Integer  -> Integer;
     /* return the indexed element in the bag */
makebag  : Integer      -> S;
     /* build a bag of one element */
"in"     : Integer, S   -> Boolean;
     /* gives true if the element is in the bag */
"<"      : S, S          -> Boolean;
">"      : S, S          -> Boolean;
"<="     : S, S          -> Boolean;
">="     : S, S          -> Boolean;
     /* subset/superset comparison operators */
"and"    : S, S          -> S;
"or"     : S, S          -> S;
     /* intersection/union operators */
make!    : * Integer     -> S;
     /* adds the included elements to the bag,
      * corresponds to (. .) */
```

Useful Types

The types `GeneralizedTime` and `UTCTime` have been defined in terms of ASN.1 as specified in X.680. It follows from their definition in X.680, together with the information about the translation rules given in this chapter, which operators are available in SDL for these types.

Constrained Types

Constrained types are mapped to sdl syntypes of the associated parent sort. Value constraints are mapped to sdl range condition.

When specifying ASN.1 value constraints, several constructs can be used that are not supported in the SDL suite, such as ALL EXCEPT,INCLUDES <subtype> and value range with MIN or MAX endpoint. Possible values for such a type are computed and mapped to syntype with range condition represented by a sequence of open and closed ranges.

Example 45 –––

```
T  ::= INTEGER ( (1..10) EXCEPT 8 )

T1 ::= INTEGER ( INCLUDES T EXCEPT (3..<6) )
```

is mapped to

```
syntype T = Integer
            constants 9 : 10, 1 : 7
endsyntype;

syntype T1 = Integer
             constants 9 : 10, 6 : 7, 1 : 2
endsyntype;
```

–––

“COMPONENTS OF” and “WITH COMPONENT” constraints are mapped by using extra inline types. If the present constraint is applied to the parent type, then the new type is generated excluding fields marked as ABSENT and including fields marked as PRESENT.

Example 46 –––

```
T  ::= SEQUENCE {
         a INTEGER,
         b IA5String
}

T1 ::=  T ( WITH COMPONENTS {
              a (-5..5),
                  b (SIZE (7))
} )
```

is mapped to

```
newtype T struct
    a  Integer;
    b  IA5String;
endnewtype;

newtype T1 struct
    a  T1_INLINE_0;
    b  T1_INLINE_1;
endnewtype;

syntype T1_INLINE_0 = Integer constants -5 : 5
endsyntype;

syntype T1_INLINE_1 = IA5String constants size(7)
endsyntype;
```

Example 47 ---

```
T  ::= SET OF BIT STRING

T1 ::= T ( WITH COMPONENT (SIZE (5)) )
```

is mapped to

```
newtype T
    Bag(Bit_string)
endnewtype;

syntype T1 = T1_INLINE_0 endsyntype;

newtype T1_INLINE_0 /*#SYNT*/
    Bag(T1_INLINE_1)
endnewtype;

syntype T1_INLINE_1 = Bit_string
    constants size (5)
endsyntype;
```

Example 48 ---

```
T ::= SET {
        a INTEGER OPTIONAL,
        b REAL OPTIONAL
}
```

```
T1 ::= T ( WITH COMPONENTS {
             a (0..<MAX) PRESENT,
             b ABSENT
} )
```

is mapped to

```
newtype T struct
   a  Integer  optional;
   b  Real  optional;
endnewtype;

newtype T1 struct
    a  T1_INLINE_0;
endnewtype;

syntype T1_INLINE_0 = Integer constants >=0
endsyntype;
```

Note:

According to ASN.1, the types T and T1 are compatible, because they are derived from each other. In SDL these are different types and values of type T can not be assigned to type T1.

ASN.1 SET OF and SEQUENCE OF types with SIZE or single value constraints are mapped to one SDL type with constraint without introducing any extra inline types.

Example 49

```
T1 ::= SEQUENCE SIZE (5..15) OF INTEGER

T2 ::= SEQUENCE ( { 1 } | {} ) OF INTEGER

T3 ::= SET (SIZE (MIN .. <100) ) OF BOOLEAN

T4 ::= SET (SIZE (15) | { ''B } ) OF BIT STRING
```

is mapped to

```
newtype T1
   String(Integer, emptystring)
   constants size (5 : 15)
endnewtype;

newtype T2
   String(Integer, emptystring)
   constants ((. .)), ((. 1 .))
endnewtype;

newtype T3
```

```
      Bag(Boolean)
      constants size (<=99)
   endnewtype;

   newtype T4
      Bag(Bit_string)
      constants ((. bitstr('') .)), size (15)
   endnewtype;
```

Extensibility

Extensibility was introduced in X.680 (1997). In ASN.1 extensibility is represented with extension markers and extension addition groups, that can be specified inside SET, SEQUENCE, CHOICE, ENUMERATED types and constraints.

Extension markers are not visible in SDL translations. All square brackets are ignored and all components from extension addition groups are translated into SDL as individual fields. All required components from extension additions, individual or from extension addition groups are mapped to optional ones.

Example 50

```
   S1 ::= SET
   {
     x [100] INTEGER,
     ... ,
     [[
        gr11 REAL
     ]],
     t BIT STRING,
     [[
        gr21 BOOLEAN OPTIONAL,
        gr22 SET OF INTEGER
     ]],
     ... ,
     y INTEGER
   }
```

is mapped to SDL

```
   newtype S1 struct
     x  Integer;
     gr11  Real  optional;
     t  Bit_string optional;
     gr21  Boolean  optional;
     gr22  S1_INLINE_1  optional;
     y  Integer;
```

```
endnewtype;

newtype S1_INLINE_1 /*#SYNT*/
   Bag(Integer)
endnewtype;
```

Note:

SDL translation removes the borders of additional groups and makes all required components optional.The semantics for assigning values to types with additional groups are: either the whole addition group ([[....]]) is absent, or it is all present unless components inside the group are optional. This is not checked in SDL tools but inconsistency will cause errors in ASN.1 encoding.

Extension markers are ignored in constraints. If both root and additional constraints are present, they are translated to the union constraint.

Example 51 ---

```
T1 ::= INTEGER ( 1..10 ^ 2..20, ... , 12 )

is mapped to SDL

syntype T1 = Integer constants 12, 2 : 10
endsyntype;

T2 ::= INTEGER (1 | 3, ... )

is mapped to SDL

syntype T2 = Integer constants 3, 1 endsyntype;
```

Information from Object Classes, Objects and Object Sets

Object classes, object and objects sets are not translated to SDL. Only types and values are translated to SDL, but it is possible in ASN.1 to use information from object classes, objects and object sets when specifying types and values. This information is translated into SDL.

ObjectClassFieldType

ObjectClassFieldType is a reference to object class and a field in that class. The translation to SDL depends on the kind of field name used.

An open type is defined if the field name references a type field, a variable type value field or variable type value set field. An open type can be any ASN.1 type. Open types are translated to Octet_string types in SDL.

Example 52 ––

```
OPERATION ::= CLASS {
   &ArgumentType,
   &arg &ArgumentType
}

T1 ::= SEQUENCE { a OPERATION.&ArgumentType }
```

is translated to SDL

```
newtype T1 struct
  a  T1_INLINE_0;
endnewtype;

syntype T1_INLINE_0 = Octet_string endsyntype;


T2 ::= OPERATION.&arg
```

is translated to SDL

```
syntype T2 = Octet_string endsyntype;
```

––

If the field name in the class references a fixed type value or fixed type value set fields, then the fixed type is used when translated to SDL.

Example 53 ––

```
OPERATION ::= CLASS {
      &ValueSet INTEGER
   }

T ::= OPERATION.&ValueSet
```

is translated to SDL

```
syntype T = Integer endsyntype;
```

––

ObjectClassFieldType with table constraint (object set constraint)

Table constraint applied to ObjectClassFieldType restricts the set of possible types or values to those specified in a column of the table. A table corresponds to an object set. The columns of the table correspond to the object class fields and the rows correspond to the objects in the set.

If the field name in ObjectClassFieldType is a type field and constrained with a table, then it is translated to a CHOICE type with fields of the types specified in the table column. The names of the fields in the choice are the same as the names of the types in the column but the first letter is changed from upper case to lower case.

Note: Field names

If the type in the field is inline then the name in the field will be an implicitly generated inline name, like t_INLINE_4.

If the field name in ObjectClassFieldType is a fixed type value or a fixed type value set, then this is translated to a constrained type where only values that are specified in the table column are permitted.

If the field name in ObjectClassFieldType is a variable type value or variable type value set field, then this is translated to a CHOICE type with types, that are constrained to have values specified in the corresponding cell in the same row of the table.

Example 54 ––

```
OPERATION ::= CLASS {
       &ArgumentType,
       &operationCode INTEGER UNIQUE,
       &ValueSet INTEGER,
       &ArgSet &ArgumentType
   }
```

The My-Operations object set

Object name	&ArgumentType	&operationCode	&ValueSet	&ArgSet
operationA	INTEGER	1	{1 | 2 | 5 .. 8}	{ 111..444 }

Object name	**&Argu-mentType**	**&opera-tionCode**	**&ValueSet**	**&ArgSet**
operationB	SET OF IN-TEGER	2	{ 2 .. 8 }	{ {1,2,3} | { 888 } }

```
C1 ::= OPERATION.&ArgumentType ( {My-Operations} )
```

is translated to SDL

```
newtype C1 choice
  integer  Integer;
  c1_INLINE_2  C1_INLINE_1;
endnewtype;

newtype C1_INLINE_1 /*#SYNT*/
   Bag(Integer)
endnewtype;
```

```
C2 ::= OPERATION.&operationCode ( {My-Operations} )
```

is translated to SDL

```
syntype C2 = Integer constants 2, 1 endsyntype;
```

```
C3 ::= OPERATION.&ValueSet ( {My-Operations} )
```

is translated to SDL

```
syntype C3 = Integer constants 2 : 8, 1, 5 : 8, 2
endsyntype;
```

```
C4 ::= OPERATION.&ArgSet ( {My-Operations} )
```

is translated to SDL

```
newtype C4 choice
  c4_INLINE_1  C4_INLINE_1;
  c4_INLINE_1  C4_INLINE_2;
endnewtype;

syntype C4_INLINE_1 = Integer constants 111 : 444
endsyntype;

syntype C4_INLINE_2 /*#SYNT*/
  Bag(Integer)
  constants ((. 888 .)), ((. 1, 2, 3 .))
endsyntype;
```

If an open type is constrained by the table for which all type settings are omitted, then it is translated to SDL Octet_string instead of an empty CHOICE type.

Example 55 ––

```
MY-CLASS ::= CLASS {
       &id INTEGER,
       &OpenType OPTIONAL
   }
```

The My-Set object set:

Object name	&id	&OpenType
object1	1	-
object2	2	-

```
S ::= SEQUENCE {
        id  MY-CLASS.&id({My-Set}),
        val MY-CLASS.&OpenType({My-Set}{@id})
}
```

is translated to SDL

```
newtype S struct
  id  S_INLINE_0;
  val  S_INLINE_2;
endnewtype;

syntype S_INLINE_0 = Integer constants 2, 1
endsyntype;

syntype S_INLINE_2 = Octet_string endsyntype;
```

––

TypeFromObject

TypeFromObject is a reference to an object and a type field in that object. This is simply translated to that type in SDL. If the field is optional in the class and not set in the object, then TypeFromObject cannot be translated.

Example 56 ––

```
OPERATION ::= CLASS {
       &ArgumentType,
           &ResultType
       }

   operationA OPERATION ::= {
          &ArgumentType INTEGER,
           &ResultType BOOLEAN
       }

   O1 ::= operationA.&ArgumentType
```

is translated to SDL

```
   syntype O1 = Integer endsyntype;


   O2 ::= operationA.&ResultType
```

is translated to SDL

```
   syntype O2 = Boolean endsyntype;
```

–––

ValueSetFromObject

ValueSetFromObject is a reference to an object and a field with a set of values in that object. This is translated to a constrained type in SDL, allowing only values from the value set.

Example 57 ––

```
   OPERATION ::= CLASS {
          &ValueSet INTEGER
       }

   operationA OPERATION ::= {
           &ValueSet { 1 | 2 | 5..8 }
      }

   V1 ::= operationA.&ValueSet
```

is translated to SDL

```
   syntype V1 = Integer constants 2, 5 : 8, 1
   endsyntype;
```

–––

ValueFromObject

ValueFromObject is a reference to an object and a field with a value in that object. This is translated to the same value in SDL.

Example 58 ––

```
OPERATION ::= CLASS {
      &operationCode INTEGER UNIQUE
   }

operationA OPERATION ::= { &operationCode 1 }

val2 INTEGER ::= operationA.&operationCode
```

is mapped to SDL

```
synonym val2 Integer = 1;
```

––

CONSTRAINED BY notation

CONSTRAINED BY is treated like a comment and is not translated to SDL.

Parameterization

Wherever a parameterized type or value is used, it is translated to SDL after all dummy references are replaced by the actual parameters. A parameterized value is also translated after all dummy references are replaced by the actual parameters.

Parameterized assignments are ignored when translating to SDL.

Example 59 ––

```
Container { ElemType, INTEGER : maxelements } ::=
       SET SIZE (0..maxelements ) OF ElemType

Intcontainer ::= Container {INTEGER, 25}
```

is first internally mapped to

```
Intcontainer ::= SET SIZE( 0..25 ) OF INTEGER
```

and then translated to SDL. Container is not translated to SDL.

––

Support for External ASN.1 in the TTCN Suite

The ASN.1 Utilities are also used by the TTCN suite if a TTCN test suite contains data types and constraints that are defined in the tables "ASN.1 Type Definitions By Reference" and "ASN.1 Constraints By Reference". For more information, see "ASN.1 External Type/Value References" on page 1188 in chapter 27, *Analyzing TTCN Documents (on UNIX)*.

Since TTCN is based on the older X.228 standard, while the ASN.1 Utilities are based on the new X.680 standard, users should be careful to use the common subset of X.680 and X.228 if an ASN.1 module is to be used in TTCN. In particular there are a number of differences:

- In `ENUMERATED` types, a value must be supplied for all values. For example:

  ```
  E ::= ENUMERATED { a, b }
  ```
  should be replaced by

  ```
  E ::= ENUMERATED { a(0), b(1) }
  ```

- X.680 offers more possibilities for specifying constraints than X.228 does. X.228 does not have the keywords `ALL`, `EXCEPT`, `UNION`, and `INTERSECTION`.

- For ASN.1 types that have components (e.g. `SET` or `SEQUENCE`), an identifier must be provided for every component (according to X.680), while in X.228 identifiers can be omitted. For example:

  ```
  S ::= SEQUENCE { INTEGER } -- valid X.228
  ```
  This is invalid according to X.680. The following should be used instead:

  ```
  S ::= SEQUENCE { field1 INTEGER }
  ```

General

- '-' (dash) in ASN.1 names is transformed to '_' (underscore), e.g. `long-name` in ASN.1 is transformed to `long_name` in TTCN.

- In general ASN.1 to TTCN translation look like pretty printing of ASN.1 modules into TTCN tables for most of the constructs, but not all of them. Some ASN.1 concepts are not supported in TTCN suite, they have to be modified during TTCN generation:

- Concepts defined in X.681, X.682 and X.683 (see “Information from Object Classes, Objects and Object Sets” on page 736, “CONSTRAINED BY notation” on page 741 and “Parameterization” on page 741)
- automatic tagging (see “Keywords substitution” on page 730)
- COMPONENTS OF Type notation (see “COMPONENTS OF Type notation” on page 733)
- selection types (see “Selection types” on page 734)
- enumerated types without numbers for enum identifiers (see “Enumerated types” on page 734)
- extensibility (see “Extensibility” on page 721)

Keywords substitution

ASN.1 generators can be configured to be sensitive to a certain number of identifiers. There is a special text file named ‘asn1util_kwd.txt’ that contains a list of identifiers and a list of their substitution during mapping. By default this file is used to configure target languages keywords substitution. It can be edited to get another functionality or another set of keywords to be replaced.

‘asn1util_kwd.txt’ should contain pairs of identifiers where the first one is the identifier from the original ASN.1 specification that will be replaced by the second identifier during generation. ‘asn1util_kwd.txt’ should conform to the following syntax:

Example 60 Configuration file syntax ––––––––––––––––––––––––––––

```
<identifier1>   <identifier1 substitution>
<identifier2>   <identifier2 substitution>
            ...
<identifierN>   <identifierN substitution>
```

––

ASN.1 Utility reads the first configuration file it finds. It searches for ‘asn1util_kwd.txt’ file first in the current folder, then in the home folder and finally in the installation. If a configuration file named ‘asn1util_kwd.txt’ is put in the home folder or in the current working folder, it will override the default configuration file from the installation. The configuration file to be used can also be specified in the command line with the '-K' option (see "Command-Line Interface" on page 697), for example,

Example 61 Configuration file specification ––––––––––––––––––––––

```
asn1util -K my_config.txt -i File.ttcn File.asn
```

–––

'asn1util_kwd.txt' is always present in the installation and it is configured to replace keywords from the SDL, TTCN, C and C++ languages: the ASN.1 identifier `name`, that is a keyword in SDL, TTCN or C++, is replaced by `name_SDL_KEYWORD`, `name_TTCN_KEYWORD` or `name_CPP_KEYWORD` to avoid syntax errors in the target languages (see "Appendix A: List of recognized keywords" on page 755). If an original ASN.1 identifier has been modified, a warning message is reported. See "ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2" on page 754).

Example 62 Keywords default substitution ––––––––––––––––––––––

For these ASN.1 definitions:

```
CASE ::= ENUMERATED { upper, lower }

T ::= SEQUENCE {
        int INTEGER,
        explicit BOOLEAN,
        case     CASE,
        signal   INTEGER
}

value1 T ::= {
        int 5,
        explicit TRUE,
        case lower,
        signal 27 }
```

With default keywords substitution file the following TTCN is generated:

```
CASE_TTCN_KEYWORD ::=
  ENUMERATED {upper(0), lower(1)}

T ::= SEQUENCE {
        int_CPP_KEYWORD INTEGER,
        explicit_CPP_KEYWORD BOOLEAN,
        case_CPP_KEYWORD CASE_TTCN_KEYWORD,
        signal_SDL_KEYWORD INTEGER
}

value1 T ::= {
  int_CPP_KEYWORD 5,
  explicit_CPP_KEYWORD TRUE,
  case_CPP_KEYWORD lower,
  signal_SDL_KEYWORD 27
}
```

A configuration file allows user to control the set of keywords to be replaced. Removing lines with SDL keywords, for example, will switch off SDL keywords sensitivity. Providing an empty configuration file will result in switching off keywords substitution completely.

Automatic tagging

If ’AUTOMATIC TAGS’ is written in the header of an external ASN.1 module, then implicit tags are inserted into SET, SEQUENCE and CHOICE types. During the TTCN generation they are inserted in the type definitions explicitly.

Example 63

```
M1
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
T ::= SEQUENCE
{
   a INTEGER OPTIONAL,
   b INTEGER DEFAULT 5
}

C ::= CHOICE
  {
    x INTEGER,
    y BOOLEAN,
    z REAL
  }
END

is translated to TTCN

SEQUENCE
{
  a [0] INTEGER OPTIONAL,
  b [1] INTEGER DEFAULT 5
}
```

```
CHOICE
{
  x [0] INTEGER,
  y [1] BOOLEAN,
  z [2] REAL
}
```

COMPONENTS OF Type notation

COMPONENTS OF Type can appear in SET or SEQUENCE field types. Instead of COMPONENTS OF Type a list of components of the referenced type is included, except extension addition components.

Example 64 ---

```
 S1 ::= SEQUENCE
{
  x INTEGER,
  g NULL,
  ... ,
  [[
     y BOOLEAN,
     z BIT STRING
  ]],
  [[
     c IA5String
  ]],
  d SET OF
    INTEGER OPTIONAL,
  ... ,
  f REAL
}

S2 ::= SEQUENCE
 {
   a IA5String,
   COMPONENTS OF S1,
   b OCTET STRING
 }

Type S2 is translated to TTCN

SEQUENCE
{
  a IA5String,
  x INTEGER,
  g NULL,
  f REAL,
  b OCTET STRING
}
```

Selection types

A selection type is mapped to the type it denotes.

Example 65 ––

```
C ::= CHOICE
 {
   a INTEGER,
   b BOOLEAN
 }

T1 ::= a < C

T1 is translated to TTCN

INTEGER

T2 ::= b < C

T2 is translated to TTCN

BOOLEAN
```

––

Enumerated types

Enumerated items can be defined using “identifier” notation or “identifier and number” notation. For “identifier” notations, implicit numbers are assigned to the identifiers according to the rules described in X.680 (1997), 19.

For TTCN, all enumeration items are generated with their corresponding numbers using “identifier and number” notation, and they are arranged according to their number values in ascending order in the generated enumeration.

Extension markers are ignored.

Example 66 ––

```
A ::= ENUMERATED { a, b, c(0), d, e(2) }

is translated to TTCN

ENUMERATED { c(0), a(1), e(2), b(3), d(4) }

B ::= ENUMERATED { a, b(3), ... , c(1) }

is translated to TTCN
```

```
ENUMERATED { a(0), c(1), b(3) }
```

Extensibility

Extensibility was introduced in X.680 (1997). In ASN.1 extensibility is represented with extension markers and extension addition groups, that can be specified inside SET, SEQUENCE, CHOICE, ENUMERATED types and constraints.

Extension markers are not visible in TTCN translation. All square brackets are ignored and all components from extension addition groups are translated into TTCN as individual fields. All required components from extension additions, individual or from extension addition groups, are mapped to optional ones.

Example 67 ---

```
S1 ::= SET
{
  x [100] INTEGER,
  ... ,
  [[
     gr11 REAL
  ]],
  t BIT STRING,
  [[
     gr21 BOOLEAN OPTIONAL,
     gr22 SET OF INTEGER
  ]],
  ... ,
  y INTEGER
}

is translated to TTCN

SET
 {
   x [100] INTEGER,
   gr11 REAL OPTIONAL,
   t BIT STRING OPTIONAL,
   gr21 BOOLEAN OPTIONAL,
   gr22 SET OF INTEGER OPTIONAL,
   y INTEGER
 }
```

Note:

TTCN translation removes the borders of additional groups and makes all required components optional. The semantics for assigning values to types with additional groups is: either the whole addition group ([[....]]) is absent, or it is all present unless components inside the group are optional. This is not checked in TTCN tools but inconsistency will cause errors in ASN.1 encoding.

```
Extension markers are ignored in constraints. If
both root and additional constraints are present,
they are translated to the union constraint.
```

Example 68 –––

```
T1 ::= INTEGER ( 1..10 ^ 2..20, ... , 12 )

is translated to TTCN

INTEGER (( 1..10 ^ 2..20 ) | ( 12 ) )

T2 ::= INTEGER (1 | 3, ... )

is translated to TTCN

INTEGER (1 | 3 )
```

–––

Information from Object Classes, Objects and Object Sets

Object classes, object and objects sets are not translated to TTCN. Only types and values are translated to TTCN, but it is possible in ASN.1 to use information from object classes, objects and object sets when specifying types and values. This information is translated into TTCN.

ObjectClassFieldType

ObjectClassFieldType is a reference to an object class and a field in that class. The translation to TTCN depends on the kind of field name used.

An open type is defined if the field name references a type field, a variable type value field or variable type value set field. An open type can be any ASN.1 type. Open types are translated to OCTET STRING types in TTCN.

Example 69 ––

```
   OPERATION ::= CLASS {
      &ArgumentType,
      &arg &ArgumentType
   }

   T1 ::= SEQUENCE { a OPERATION.&ArgumentType }
```

is translated to TTCN

```
   SEQUENCE { a  OCTET STRING }

   T2 ::= OPERATION.&arg
```

is translated to TTCN

```
   OCTET STRING
```

––

If the field name in the class references a fixed type value or fixed type value set fields, then the fixed type is used when translated to TTCN.

Example 70 ––

```
   OPERATION ::= CLASS {
         &ValueSet INTEGER
      }

   T ::= OPERATION.&ValueSet
```

is translated to TTCN

```
   INTEGER
```

––

ObjectClassFieldType with table constraint (object set constraint)

A table constraint applied to ObjectClassFieldType restricts the set of possible types or values to those specified in a column of the table. A table corresponds to an object set. The columns of the table correspond to the object class fields and the rows correspond to the objects in the set.

If the field name in ObjectClassFieldType is a type field and constrained with a table, then it is translated to a CHOICE type with fields of the types specified in the table column. The names of the fields in the choice are the same as the names of the types in the column but the first letter is changed from upper case to lower case.

Note:

If the type in the field is inline then the name in the field will be an implicitly generated inline name, like t_INLINE_4.

If the field name in ObjectClassFieldType is a fixed type value or a fixed type value set, then this is translated to a constrained type where only values that are specified in the table column are permitted.

If the field name in ObjectClassFieldType is a variable type value or variable type value set field, then this is translated to a CHOICE type with types, that are constrained to have values specified in the corresponding cell in the same row of the table.

Example 71 ––

```
OPERATION ::= CLASS {
       &ArgumentType,
       &operationCode INTEGER UNIQUE,
       &ValueSet INTEGER,
       &ArgSet &ArgumentType
   }
```

The My-Operations object set:

Object name	**&ArgumentType**	**&operationCode**	**&ValueSet**	**&ArgSet**
operationA	INTEGER	1	{1 | 2 | 5 .. 8}	{ 111..444 }
operationB	SET OF INTEGER	2	{ 2 .. 8 }	{ {1,2,3} | { 888 } }

```
C1 ::= OPERATION.&ArgumentType ( {My-Operations} )
```

is translated to TTCN

```
CHOICE {
  integer INTEGER,
  c1_INLINE_2 SET OF INTEGER
}
```

```
C2 ::= OPERATION.&operationCode ( {My-Operations} )
```

is translated to TTCN

```
INTEGER ( 1 | 2 )
```

```
C3 ::= OPERATION.&ValueSet ( {My-Operations} )
```

is translated to TTCN

```
INTEGER ( ( 1 | 2 | 5..8 ) | ( 2..8 ) )
```

```
C4 ::= OPERATION.&ArgSet ( {My-Operations} )
```

is translated to TTCN

```
CHOICE {
  c4_INLINE_1 INTEGER ( (111..444) ),
  c4_INLINE_2 SET OF INTEGER ( ( {1,2,3} | {888} ) )
}
```

If an open type is constrained by the table for which all type settings are omitted, then it is translated to TTCN OCTET STRING instead of an empty CHOICE type.

Example 72 ---

```
MY-CLASS ::= CLASS {
       &id INTEGER,
       &OpenType OPTIONAL
   }
```

The My-Set object set:

Object name	&id	&OpenType
object1	1	-
object2	2	-

```
S ::= SEQUENCE {
        id  MY-CLASS.&id({My-Set}),
        val MY-CLASS.&OpenType({My-Set}{@id})
}
```

is translated to TTCN

```
SEQUENCE {
  id    INTEGER ( 1 | 2 ),
  val   OCTET STRING
}
```

TypeFromObject

TypeFromObject is a reference to an object and a type field in that object. This is simply translated to the that type in TTCN. If the field is optional in the class and not set in the object, then TypeFromObject cannot be translated.

Example 73 –––

```
OPERATION ::= CLASS {
       &ArgumentType,
       &ResultType
   }

operationA OPERATION ::= {
      &ArgumentType INTEGER,
       &ResultType BOOLEAN
   }

O1 ::= operationA.&ArgumentType
```

is translated to TTCN

```
INTEGER

O2 ::= operationB.&ResultType
```

is translated to TTCN

```
BOOLEAN
```

–––

ValueSetFromObject

ValueSetFromObject is a reference to an object and a field with a set of values in that object. This is translated to a constrained type in TTCN, allowing only values from the value set.

Example 74 –––

```
OPERATION ::= CLASS {
       &ValueSet INTEGER
   }

operationA OPERATION ::= {
       &ValueSet { 1 | 2 | 5..8 }
  }

V1 ::= operationA.&ValueSet
```

is translated to TTCN

```
INTEGER ( 1 | 2 | 5..8 )
```

ValueFromObject

ValueFromObject is a reference to an object and a field with a value in that object. This is translated to the same value in TTCN.

Example 75 ---

```
OPERATION ::= CLASS {
      &operationCode INTEGER UNIQUE
   }

operationA OPERATION ::= { &operationCode 1 }

val2 INTEGER ::= operationA.&operationCode
```

is translated to TTCN

```
val2 of type INTEGER equal to 1
```

CONSTRAINED BY notation

CONSTRAINED BY is treated like a comment and is not translated to TTCN.

Parameterization

Wherever a parameterized type or value is used, it is translated to TTCN after all dummy references are replaced by the actual parameters. A parameterized value is also translated after all dummy references are replaced by the actual parameters.

Parameterized assignments are ignored when translating to TTCN.

Example 76 ---

```
Container { ElemType, INTEGER : maxelements } ::=
       SET SIZE (0..maxelements ) OF ElemType

Intcontainer ::= Container {INTEGER, 25}
```

is mapped to TTCN

```
SET SIZE ( 0..25 ) OF INTEGER
```

Error and Warning Messages

This section contains a list of the error and warning messages in the ASN.1 Utilities. Each message has a short explanation and, where applicable, a reference to the appropriate section of the recommendations X.680,X.681,X.682, X.683 or Z.105.

Some messages include a reference to the object that is the source of the diagnostic. These messages adhere to the format adopted in the SDL suite. See chapter 19, *SDT References* for a reference to this format and for examples.

`WARNING 2000 Unknown option '#1'`

This warning message indicates that the ASN.1 Utilities were started with an unknown option. See "Command-Line Interface" on page 697 for an overview of the valid options.

`WARNING 2001 No #1 specified after '#2' option`

This warning message indicates that the ASN.1 Utilities were started with an illegal combination of options. See "Command-Line Interface" on page 697 for an overview of the valid options.

For example,

```
asn1util -s -r MyModule.asn
```

In this case no output file for sdl generation is specified after '-s' option.

`ERROR 2002 Too many errors`

This error message indicates that the maximum number of errors was reached when analyzing an ASN.1 module. The analysis has been aborted by the ASN.1 Utilities.

`ERROR 2003 Multiple #1 paths`

This warning message indicates an incorrect usage of the options of the ASN.1 Utilities.

For example,

```
asn1util -Tdir1 -Tdir2 -s MyModule.pr MyModule.asn
```

Multiple target directories provide a warning message

`WARNING 2004 Option missing`

This warning message indicates that no option is specified after dash.

`ERROR 2005 Can not open #1`

This error message indicates that an error occurred when the ASN.1 Utilities attempted to open a file. Modify, if necessary, the file protection and try to run the ASN.1 Utilities again. If the error persists, contact Telelogic Customer Support. Contact information for Telelogic Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

For example,

```
 asn1util -Tdir -i MyModule.ttcn MyModule.asn
```

This command line can cause an error message "Can not open air/MyModule.ttcn" if there is no target directory 'dir' in the catalogue from which asn1util is called.

`ERROR 2006 Illegal characters in bstring`

This message indicates that an ASN.1 binary string item (used in BIT STRING and OCTET STRING) contains illegal characters. The only characters allowed are `0', `1' and white space characters. (X.680: 9.9)

`ERROR 2007 Illegal characters in hstring`

This message indicates that an ASN.1 hexadecimal string item (used in BIT STRING and OCTET STRING) contains illegal characters. The only characters allowed are `0'-'9', `A'-'F' and white space characters. (X.680: 9.10)

For example: 'F30C 973D'H is a valid hexadecimal string item.

`ERROR 2008 'H' or 'B' expected`

This error message indicates that an ASN.1 BIT STRING or OCTET STRING value is not ended with a `B or an `H. (X.680: 9.9 and 9.10).

For example: '0110'B or '1AFC'H are valid values for BIT STRING and OCTET STRING, '01110' is illegal.

`ERROR 2009 Unclosed #1 string`

This error is reported when there is no closing apostrophe at the end of string

`WARNING 2010 Unknown token '#1'`

This warning indicates a syntax error in the ASN.1 module.

`ERROR 2011 Syntax error`

This message indicates a syntax error in the ASN.1 module with syntax from standard X.680-X.683. This could be caused by a misspelling. It could also be caused by X.228 constructs that are not part of X.680.

`ERROR 2012 Out of memory`

This message indicates that the ASN.1 Utilities ran out of memory. Try to make the ASN.1 module smaller or supply more memory. If the error persists, contact Telelogic Customer Support. Contact information for Telelogic Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

`WARNING 2013 No semantic support for '#1'`

This warning indicates that an ASN.1 construct is used that is not supported by the ASN.1 Utilities. The construct will be ignored by the ASN.1 Utilities.

`ERROR 2014 Export-file '#1' corrupt`

This message indicates that the export file format of an ASN.1 module was corrupt or unknown. This error should normally not occur. Contact Telelogic Customer Support. Contact information for Telelogic Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

`ERROR 2015 Old ASN1,#1`

This message indicates that an ASN.1 construct of the older X.228 recommendation is used that has been superseded in the X.680 Recommendation.

For example:

```
S ::= SEQUENCE { INTEGER }
```

is old ASN.1. Correct X.680 ASN.1 is:

```
S ::= SEQUENCE { field1 INTEGER }
```

`ERROR 2016 Recursive expansion of COMPONENTS OF in type #1`

This error message indicates that the ASN.1 type uses directly or indirectly COMPONENTS OF itself.

`ERROR 2017 Recursive #1`

This message indicates that the ASN.1 construct is recursively defined.

For example: `T1::=T2, T2::=T1; T::=SET OF T`, or v T ::= v

`ERROR 2018 Recursive #1 constraint`

This error message indicates that type being constrained is recursively used in applied constraint.

For example: `I ::= INTEGER ( 1 .. 10 | INCLUDES I)`

`ERROR 2019 Field '#1' should be initialized by #2`

This message is reported when you assign wrong kind of value for the field in the object, for example when you try to assign a value for the type field in the object

`ERROR 2020 Value for `#1' can not be #2`

This error message indicates a semantic error in the ASN.1 module.

For example, `T ::= BIT STRING { a(-1)}` causes the error “Value for 'named bit' can not be negative”

`WARNING 2021 Construct '#1' has no mapping in SDL`

This warning indicates that an ASN.1 construct is used that can not be mapped to SDL.

For example:

```
S1 SEQUENCE ::= { }

          -- empty SEQUENCE/SET

     s SEQUENCE { a INTEGER OPTIONAL } ::= {}

            -- value for SEQUENCE/SET without components
```

`ERROR 2022 Ambiguous reference, symbol '#1' imported more than once`

A value is used that is imported more than once. Use an external value reference to specify unambiguously the module of the value that you want to use.

`ERROR 2023 Multiple definition of #1`

This error message appears, when the same identifier appears more then once on the right side of assignment.

For example, `X ::= INTEGER, X ::= SET OF REAL`

`ERROR 2024 Exported symbol #1 not defined`

This error message is reported when symbol is exported, but it is neither defined in the module nor imported to it

`ERROR 2025 Ambiguous export, symbol #1 is imported more than once`

This error message indicates that it is impossible to decide which symbol to export, because two symbols with the name #1 are imported to the module

`ERROR 2026 Ambiguous export, symbol #1 is defined and imported`

This error message indicates that it is impossible to decide which symbol to export, because symbol #1 is defined in the module and imported to it at the same time

`ERROR 2027 Nothing known about module #1`

This message appears when module referenced from imports clause does not exist. You should specify all modules from which symbols are imported to the analyzed module in the same command line, otherwise it is impossible to perform global semantic analysis

`ERROR 2028 Import from empty module #1`

This message appears when importing symbols from a module, that does not contain any definitions

ERROR 2029 Module does not export symbols

This message appears when you are trying to import symbols from module with empty export: “EXPORTS ;”

ERROR 2030 Imported symbol #1 is not exported from module #2

This message appears when symbol #1 is present in imports from module #2 clause, but it is not exported from #2. `"EXPORTS ;"` indicates that nothing is exported, while empty exports clause indicates that all definitions are exported from the module.

ERROR 2031 Imported symbol #1 is not defined in module #2

This error situation occurs when symbol is imported from module that exports all, but symbol is not defined there

ERROR 2032 Ambiguous import, symbol #1 imported more than once to module #2

This indicates that all symbols are exported from module #2, but it is impossible to import symbol #1 from module #2 because symbol #1 is imported more than once to #2. The symbols have the same name, but defined in different modules.

ERROR 2033 Ambiguous import, symbol #1 defined and imported to module #2

This indicates that all symbols are exported from module #2, but it is impossible to import symbol #1 from module #2 due to ambiguity between symbol #1defined in module #2 and symbol #1 imported to module #2.

For example;

```
M1 DEFINITIONS ::= BEGIN
   IMPORTS a FROM M2;
END

M2 DEFINITIONS ::= BEGIN
   IMPORTS a FROM M3;
   a INTEGER ::=5
END

M3 DEFINITIONS ::= BEGIN
   EXPORTS a;
  a BOOLEAN ::= TRUE

END
```

In the above case you can not import a to M1, although a is exported from M2.

`ERROR 2034 Multiple declaratiom of module name #2`

Module name shall appear only once in IMPORTS clause.

For example

IMPORTS a , b FROM X c FROM X; is wrong ASN.1 declaration

`ERROR 2035 Recursive import for #1`

This error message is reported, for example, when module A imports T from B, and B imports T from A at the same time

`ERROR 2036 Multiple occurance of #1 ’#2’ in #3`

This error is reported when some types are defined incorrectly - they have the same identifier, for example enumeration can not have the same identifiers, named number list for INTEGER type can not have the same identifiers in the list, #2 is a string

`ERROR 2037 Multiple occurance of #1 #2 in #3`

The same class of error as ERROR 2036 above, but #2 is an integer value.

`ERROR 2038 External references are not allowed`

When imports clause looks like “IMPORTS ;”,no external references are allowed from the module (X.680, 10.14, d), NOTE 2)

`ERROR 2039 Referenced #1 ’#2’ not defined`

This error is reported when you use reference that is not assigned value or type anywhere.

`ERROR 2040 Value of type #1 needed`

This error message indicates that value does not correspond to the type. For example `x INTEGER ::= TRUE` - this results in an error “Value of type INTEGER needed”

`ERROR 2041 #1 type needed after COMPONENTS OF`

The type after in COMPONENTS OF expansion should be either SET or SEQUENCE, and it should be the same as the type to which it is extracted.

For example SET { a INTEGER, b COMPONENTS OF T }, where T is SEQUENCE type is wrong usage of COMPONENTS OF notation(X.680, 22.4, 24.2)

`ERROR 2042 Field names in type after COMPONENT OF already declared`

After performing the COMPONENTS OF transformation, all field names should be distinct.

For example, type S1 is wrong (it has two fields named ’a’)

```
S ::= SET { a INTEGER, b REAL }
S1 ::= SET { a SET OF IA5String, COMPONENTS OF S }
```

`ERROR 2043 #1 type needed`

This error is reported when type in selection type is not choice.

For example x < INTEGER does not satisfy that requirement

`ERROR 2044  No alternative named #1 in Choice type`

This error is reported when type notation is “#1 < type”, type is a CHOICE type, but it does not have alternative named #1

`ERROR 2045 Too many components`

This error message appears when you are trying to assign extra components, which are not defined in the type, when specifying the value of SET or SEQUENCE

`ERROR 2046 No such field ’#1’ in #2 type`

This error indicates that type #2 does not have field named #1, but you are trying to assign it a value.

`ERROR 2047 Missing values for non-optional #1 fields : #2`

This message indicates that not all required #1 components have been initialized in the value, and #2 is the list of names of fields, for which values are missing. The example S ::= SET { a INTEGER, b REAL, c

NULL OPTIONAL }, s S ::= { a 57 } results in error message “Missing values for non-optional SET fields: ’b’”.

ERROR 2048 More than one #1 for the field ’#2’

This error occurs when you are trying to assign more then one component to one field.

For example

```
T ::= SET { a IA5String , b NULL }
t T ::= { a "val1", b NULL, a "val2" }
```

ERROR 2049 Nothing known about bit named ’#1’

This error is reported when bitstring value contains identifier that is not declared in the correspondent type definition

ERROR 2050 Value for #1 should be #2

If you specify table for the value of IA5String, TableColumn should be in the range from 0 to 7, if this constraint is violated then the above error message is reported

ERROR 2051 Type is required to be derived from #1

This error indicates that type used in SubtypeConstraint is not derived from the type being constrained and thus does not satisfy X.680,45.3.2

ERROR 2052 Can not apply #1 constraint to #2 type

Not all constraints can be applied to every type, X.680, Table 6 describes which constraints can be applied to which types, if the requirements declared in Table 6 are not satisfied, the above error message is reported

ERROR 2053 There shall be at most one #1

Two presents constraints are not allowed when constraining a CHOICE type.

For example:

```
C ::= CHOICE { a T1 , b T2 } (WITH COMPONENTS {a PRESENT,
b PRESENT} )
```

causes the error message

`ERROR 2054 Wrong value : out of constraint`

This error is reported when value does not correspond to the constraint applied to the type.

For example: `x INTEGER (1..10) ::= -1`, x is out of constraint

`ERROR 2055 The same tags for #1 components`

This error message indicates that type does not correspond to the requirements for distinct tags specified in X.680, 22.5, 24.3, 26.2; If you use AUTOMATIC TAGS in the module, requirement for distinct tags will always be satisfied if automatic tagging has been applied

`ERROR 2056 OBJECT IDENTIFIER value should have at least two components`

`x OBJECT IDENTIFIER ::= { iso }` is wrong object identifier value because encode/decode functions require at least two components for object identifier value

`WARNING 2057 Construct #1 has no mapping in SDL`

This warning is reported if no mapping to sdl exist but it does not prevent further code generation.

`ERROR 2058 Construct #1 has no mapping in SDL`

This error indicates that no mapping to sdl exist and is fatal for further code generation.

`WARNING 2059 Value given for #1 component`

This warning indicates that a value has been given to an optional or default component of an ASN.1 SEQUENCE or SET type. Values for optional and default components cannot be translated to SDL.

`WARNING 2060 Constraint could have been extended when mapped to sdl`

This warning indicates that constraint transformation has been applied when mapping complex ASN.1 constraints to sdl but the sdl type can allow more values than the ASN.1 type. This can occur when there is no exact mapping of ASN.1 constraints.

`ERROR 2061 INTERNAL ERROR in #1`

This message indicates an error in the implementation in the utilities. Please send a report to Telelogic Customer Support, especially if the error can be reproduced as the only error message of an analysis. Contact information for Telelogic Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*

`ERROR 2062 Code generation : #1`

Error in the generation of SDL, TTCN or encode/decode output.

`WARNING 2063 Too big exponent`

Exponent in a real value is too big to translate to SDL. This warning message is shown if the exponent is bigger than 1000 or less than -1000.

`WARNING 2064 Duplicate synonym name, this synonym will not be mapped to SDL`

This message indicates that there are synonym name clashes between named numbers and named bits from INTEGER and BIT STRING types and ASN.1 values if they all will be mapped to SDL (see "Integer" on page 713 and "Bit String" on page 710), and in order to avoid errors only one synonym will be mapped, others are ignored.

`ERROR 2065 Number #1 is already assigned to previously defined enumeration item`

This error message is reported when NamedNumber alternative is used in an enumerated type definition in an addition enumeration after extension marker and the number #1 has already been assigned to identifier from root enumeration, for example   A ::= ENUMERATED {a,b, ... , c(0)} First corresponding numbers are assigned to identifiers in root enumeration, and then in additional enumeration. The above case is invalid, since both ’a’ and ’c’ are equal to 0.

`ERROR 2066 Value for the field ’#1’ needed #2`

This message indicates that value for the field ’#1’ is missing, but it should be present in #2

`ERROR 2067 #1 omitted in #2`

Indicates that #1 is omitted, but it should be present in #2

`ERROR 2068 #1 should reference #2`

This message indicates that a field name references an object class field that is not allowed to be referenced.

`ERROR 2069 Wrong object specification`

This message indicates that an object specification is incorrect

`ERROR 2070 #1 of class #2 needed`

This message indicates that an object or object set does not match the governing object class specified in object or object set's definitions

`ERROR 2071 Wrong defined syntax`

This message indicates that an error in defined syntax for the object definition

`ERROR 2072 #1 can not be used in object set specification`

This message indicates that an illegal construct is used in the object set specification

`ERROR 2073 #1 in the field '#2' is not specified in the #3`

This message is reported when information, for example, type or value, is extracted from object field that has not been initialized in the object. This can occur when the field is optional or default in the object class.

`WARNING 2074 #1 is not supported in the encoders / decoders`

This warning message is reported when ASN.1 notation is used that the encoder / decoder library cannot support

`ERROR 2075 #1 can be used only for #2`

This message is reported when #1 is used in a component relation constraint but is not allowed to be used in that context.

`ERROR 2076 #1`

This is used for several different messages concerning component relation constraint, each message is listed and explained below:

`Referenced component should refer to the same object class as the referencing one`

This message indicates that the referenced and referencing components in a component relation constraint do not stem from the same object class.

`Only fixed type value fields are allowed to be specified in a referenced component`

This message indicates that a referenced component in a component relation constraint is not a fixed type value field, for example

```
SET {
      a MY-CLASS.@id ({My-set}),
      b MY-CLASS.@TypeField ({My-set})
    }
```

For the field a, @id should reference fixed type value field in class 'MY-CLASS'

`Only values of INTEGER types can be used as component relation identifiers";`

This message is reported when a referenced fixed type value field is not an INTEGER, only INTEGERS are supported.

```
SET {
      a MY-CLASS.@id ({My-set}),
      b MY-CLASS.@TypeField ({My-set})
    }
```

In the example above @id should be derived from an INTEGER.

`Wrong referenced component`

This message indicates that a wrong type of component is referenced in a component relation constraint, for example not using ObjectClass-FieldType notation.

`Referenced components should be constrained by the same object set as the referencing one`

This message indicates that the referenced and referencing component are not constrained with the same object sets.

`ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2`

This message is reported when an ASN.1 identifier is a keyword in one of the target languages and will be changed according to the keywords configuration file during mapping for avoiding syntax errors in the target languages.

`ERROR 2078 Module #1 has got name clashes within joined modules group '#2'`

This message is reported when an ASN.1 definition name causes name clashes within joined SDL package and will be prefixed by the original ASN.1 module name during mapping to avoid errors in SDL (see "Joining modules" on page 704).

Restrictions

The ASN.1 Utilities handle all constructs of ASN.1 as defined in ITU-T recommendations X.680, X.681, X.682, X.683, X.690, X.691. There is no support for features defined in the old ASN.1 version X.208 that have been superseded in X.680.

For a list of restrictions see "ASN.1 Utilities" on page 29 in chapter 2, *Release Notes, in the Release Guide*.

Appendix A: List of recognized keywords

By default target language keywords are recognized among ASN.1 identifiers and a postfix `'_<language>_KEYWORD'` is added at the end of the identifier when SDL (`<language> = SDL`), TTCN (`<language> = TTCN`) or C (`<language> = CPP`) is generated. This appendix describes lists of supported keywords for all supported target the target languages.

SDL keywords

```
active, adding, all, alternative, and, any, as, atleast,
axioms, block, break, call, channel, choice, comment,
connect, connection, constant, constants, continue, cre-
ate, dcl, decision, default, else, endalternative, end-
block, endchannel, endconnection, enddecision, endgener-
ator, endmacro, endnewtype, endoperator, endpackage,
endprocedure, endprocess, endrefinement, endselect, end-
service, endstate, endsubstructure, endsyntype, endsys-
tem, env, error, export, exported, external, fi, final-
ized, for, fpar, from, gate, generator, if, import, im-
ported, in, inherits, input, interface, join, literal,
literals, macro, macrodefinition, macroid, map, mod,
nameclass, newtype, nextstate, nodelay, noequality,
```

none, not, now, offspring, operator, operators, optional, or, ordering, out, output, package, parent, priority, procedure, process, provided, redefined, referenced, refinement, rem, remote, reset, return, returns, revealed, reverse, save, select, self, sender, service, set, signal, signallist, signalroute, signalset, size, spelling, start, state, stop, struct, substructure, synonym, syntype, system, task, then, this, timer, to, type, use, via, view, viewed, virtual, with, xor

TTCN keywords

ACTIVATE, AND, BITSTRING, BIT_TO_INT, BY, CANCEL, CASE, COMPLEMENT, CP, CREATE, DO, DONE, ELSE, ENC, ENDCASE, ENDIF, ENDVAR, ENDWHILE, F, FAIL, fail, GOTO, HEXSTRING, HEX_TO_INT, I, IF, IF_PRESENT, INCONC, inconc, INFINITY, INT_TO_BIT, INT_TO_HEX, IS_CHOSEN, IUT, LT, min, MOD, ms, MTC, NOT, ns, OMIT, OR, OTHERWISE, P, LENGTH_OF, none, NUMBER_OF_ELEMENTS, OCTETSTRING, OBJECTIDENTIFIER, PASS, pass, PDU, PERMUTATION, ps, PTC, R, READTIMER, REPEAT, REPLACE, RETURN, RETURNVALUE, R_Type, s, START, STATIC, SUPERSET, SUBSET, THEN, TIMEOUT, TIMER, TO, UNTIL, us, UT, VAR, WHILE

C++ keywords

bool, catch, class, const_cast, delete, dynamic_cast, explicit, false, friend, inline, __multiple_inheritance, mutable, namespace, new, operator, private, protected, public, reinterpret_cast, __single_inheritance, static_cast, template, this, throw, true, try, typeid, typename, using, virtual, __virtual_inheritance, xalloc

auto, asm, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while