GENERAL INFORMATION
For further information, comments and suggestions about EDT package please contact:

Stan BUDKOWSKI  
INT - Institut National des Télécommunications  
Software-Networks Department  
9, rue Charles Fourier, 91011 EVRY, FRANCE  
Tel: +33 (1) 60 76 47 20  
Fax: +33 (1) 60 76 47 11  
E_mail: stan@int-evry.fr

Note: The following persons participated in the EDT development:


UNIX is a registered trademark of Unix Systems Labs  
MS-DOS is a registered trademark of Microsoft Corporation  
OS2 is a registered trademark of IBM Corporation  
Windows NT is a registered trademark of Microsoft Corporation  
FLEXlm is a registered trademark of Globtrotter Software, Incorporated  
All other product names mentioned herein are the trademarks of their respective owners.
# TABLE OF CONTENTS

1. INTRODUCTION ........................................................................................................... 5  
2. ESTELLE RESTRICTIONS AND EXTENSIONS ............................................................ 9  
3. SYSTEM REQUIREMENTS AND INSTALLATION ....................................................... 10  
   3.1 License environment .............................................................................................. 11  
4. PRIMITIVE FUNCTIONS AND PROCEDURES IN EDT .......................................... 13  
   4.1 Estelle ‘primitives’ delivered with the EDT package ............................................ 16  
5. ERROR PROCESSING IN EDT .................................................................................... 16  
   5.1. Pascal run-time errors ......................................................................................... 16  
   5.2. Estelle run-time errors and warnings ................................................................. 17  
REFERENCES .................................................................................................................. 19  
APPENDIX 1 - QUALIFYING COMMENTS .................................................................. 20  
APPENDIX 2 - STANDARD I/O ESTELLE PRIMITIVES ............................................ 23
1. INTRODUCTION

This document presents a set of tools initially developed by Bull S.A (the Estelle Translator in co-operation with MARBEN) and further extended by the INT. Several new tools whose development have been partially founded by FRANCE TELECOM (CNET) have been aided. The tools allow the development of reliable implementations of systems specified in Estelle.

Estelle is a Formal Description Technique (FDT) proposed by ISO. The work on Estelle within ISO began in 1981. The present status of Estelle is that of the International Standard. The formal syntax and semantics of Estelle are described in ISO 9074 document [1] available from the ANSI Secretariat or from national organisations for standardisation.

Estelle is a specification language that can be used for the formal description of distributed, concurrent processing systems. In particular, Estelle can be used to describe the services and protocols of Open System Interconnection architecture defined in the ISO 7498 standard.

An Estelle specification is a hierarchically structured collection of communicating nondeterministic state automata (transition systems) whose transitions' actions are given in the form of Pascal [2] statements with some extensions and restrictions.

The EDT set of tools is composed of:

- An **Estelle Compiler (Ec)**, which checks an Estelle specification for static errors and translates it, if no errors are present, into the Intermediate Form (IF) and then into the C language source code. The Estelle Compiler is in fact composed of two tools: a Translator and a C-code Generator;

- An **Estelle Simulator/Debugger (Edb)**, which allows the validation of Estelle specifications with respect to dynamic errors. It provides also facilities to make performance prediction;

- A **Universal Generator (UG)**, which permits to:
  - generate from a given Estelle specification a modified one in which some selected module bodies are replaced by so-called universal bodies (test drivers')
  - generate from (split) an Estelle specification containing 'n' subsystems (modules attributed 'systemprocess' or 'systemactivity'):
    - 'n' Estelle specifications each containing one of the subsystems and a special interface (called also "switch board" to the external word,
    - a specification dependent parts of the RPC-client ('supervisor') and RPC-server C programs
  - generate from an intermediate form the corresponding Estelle specification (decompiler)

Each particular function of the Universal Generator (UG) may be viewed as a separate tool: **Test Drivers' Generator, Distributed Specification Generator (splitter), Decompiler**.

- An **Estelle Event/State Tables Generator (Edoc)**, which generates Event/State tables for the designated bodies within the specification.

- An **Estelle Browser** (Hld), which enables the user to generate a so-called high-level-design (hld) view of the specification
- A **Estelle Pretty Printer** (Epp), which permits to generate from an Estelle Specification an equivalent one but in a ‘pretty’ form,

All the above tools can be invoked directly under UNIX or from an Xwindow interface called XEDT (see XEDT User Reference Manual).

An **Estelle Graphic Editor** for authoring and viewing graphical Estelle specification documents is an optional part of the EDT package. The graphical Estelle language (Estelle/GR) used by the editor is the non-standard counterpart of the currently standardised textual language (Estelle/PR). Its definition is highly influenced by the SDL/GR language. The Estelle Graphic Editor provides a generator of Estelle/PR from Estelle/GR. Support for translating Estelle/PR to Estelle/GR is also provided.

The majority of tools were specified, and some early prototypes developed, within the framework of the European ESPRIT project N° 410 -- SEDOS (Software Environment for Distributed Open Systems). The development of Test Drivers' Generator (Utdg) was partially supported by ESPRIT project N° 5341 -- OSI 95. However, much additional work was done after that projects ended, with an emphasis on extending the tools' functionality and on testing their reliability. This additional work was partially supported by FRANCE TELECOM (CNET).

All tools are written in C language. They run under UNIX on SUN-Sparc (under SunOS 4.1.x and SOLARIS 2.x), HP 9000/700 (under HP-UX 10.x) and PC (under LINUX 2.x) computers.

The set of tools presented here has been used for the specification and development of many protocols, including FTAM, OSI-TP, CMIP, TP0, TP2 and XTP (ESPRIT-OSI95 project) protocols.

More than 40 Universities and Research Centres around the world (Australia, Brazil, Canada, Finland, France, Germany, Poland, Romania, Sweden, South Africa, and USA) are licensed to use the EDT tools. It has also had some experimental application in industry.

The overall EDT architecture is depicted on Figure 1.

![Fig. 1 Overall architecture of EDT tools](image-url)
The Figure 2 gives more information about the ways by which the executable code for the simulator/debugger and for prototype implementation can be obtained from an Estelle specification. It shows, in particular, the use of the CPP pre-processor (a part of the general C language environment), the possibility of using User Libraries and the fact that the same C code generated for simulation/debugging is reused when a prototype implementation is automatically generated. The validation efforts on Estelle specifications therefore have a direct impact on the correctness of the implementation.

![Figure 2. Generation of the executable code for the simulation and implementation](image)

The use of the CPP pre-processor is necessary when the Estelle specification is split into several files. The Estelle language accepted by EDT is in fact extended to accept the same directives as used within the C environment (e.g., `#include<filename>`) to enable a multi-file specification. The main part of an
Estelle specification has to have the extension '.stl' (e.g., `<source_file_name>.stl`). When the Estelle Compiler (Ec) is properly invoked (with the -I option) the CPP pre-processor is automatically invoked and a file `<source_file_name>.cpp` is generated, which contains a complete Estelle specification. All files specified by #include directives within `<source_file_name>.stl` ('spec.stl' in Fig. 2) file are included in this file and the new file is named `'<source_file_name>.cpp'`. This `<source_file_name>.cpp` file is checked by the Estelle Translator against any static errors. In addition to the lexical and syntactical errors checking, the type checking is also performed.

If no errors have been detected and the Ec was invoked with the appropriate options, a file `<source_file_name>.if` is generated which contains the Estelle specification in so-called intermediate form (IF) representation.

The C-code generator transforms the intermediate form file generated by the Estelle Translator into several files (with extensions '.c', '.h') which represents in C language all objects of the initial Estelle specification. The C-code generator may be invoked manually once the `<source_file_name>.if` file has been created, or it is automatically invoked after the Estelle Translator, provided the Ec compiler was invoked with the appropriate options and provided no errors have been detected during the translation phase (the `<source_file_name>.if` file has been created). The C-code generated is a so-called traditional C¹, in contrast to the ANSI-C.

Besides the above mentioned files (with extensions '.c' and '.h') the C-code generator may also create a makefile (`<source_file_name>.mk`) which enables the automatic creation of a file containing the executable code either for the simulation/debugging (`<source_file_name>.edb`) or for prototype implementation (`<source_file_name>`). What is finally created depends on files which are linked to the code generated by the C-code generator and compiled by the C compiler. Several files to be linked are specified automatically within the generated makefile following the options with which the Ec compiler was invoked. This is, of course, not the case of the User Libraries which have to be created specifically for the Estelle specification actually processed. In particular, the User Libraries have to be created when, within an Estelle specification, some functions and/or procedures are declared primitive (see Section 4 for some guidelines). A number of pre-defined primitives are delivered with the EDT package (see Appendix 2).

The makefile may be either executed manually (UNIX command `make -f <source_file_name>.mk`) or its execution may be automatically chained after the C-code generator is completed. In the current EDT version the automatic execution of the makefile takes place only when the Edb is invoked. Within the Edb invocation an appropriate option enables the user to specify the User Libraries to be automatically linked to the C-code generated. Please consult the Estelle Simulator/Debugger Reference Manual for details.

The EDT is an open toolset. This means that other tools, designed by the user, may be added. The opening is provided by offering the user a programming interface to the intermediate form (IF). For example, this permits the user to design and glue his own C++ code generator without being forced to redesign the Estelle Translator.

The IF programming interface definition and the guidelines for its use are included in the document "Intermediate Form: Utilization principles".

The implementation kernel delivered with EDT (automatically linked to the generated C-code) offer a prototype implementation for Unix based systems. User defined kernels for the other operating systems e.g., MS-DOS, OS2, Windows-NT can also be used, however. The makefile generated by the Estelle Compiler (Ec) must be modified to change the implementation kernel file names to be linked.

To enable a User to define his own implementation kernels the interface between the C-code generated and implementation kernels is defined (see the Reference Manual for Estelle-to-C compiler).

1B. Kernighan and D. Ritchie, 'The C programming language', Prentice Hall, 1978
2. ESTELLE RESTRICTIONS AND EXTENSIONS

The present version of EDT tools accepts Estelle as defined in [1] with some restrictions and extensions.

Restrictions

The language accepted by EDT tools is restricted in the following way:

- Integer values are limited to the range -2147483648 .. 2147483647. Therefore an integer constant (constant literal value) can only be declared within these limits.

- The length of character items (identifiers, strings, etc.) is limited, but the exact limitation is not known. Items up to 1024 characters have been tested and are processed. All characters of such items are meaningful for the analyzer.

- All elements of <type> within a "set of <type>" definition must have their ordinal numbers in the closed interval 0 to 255, e.g., if "set of (a, b, c)" is used, then all three numbers ord(a), ord(b) and ord(c) ("ord" is a Pascal required function) must be in this interval.

- Each constant C appearing in a set value "[... C ...]" (defined by the Pascal "set-constructor") must have its ordinal number (ord(C)) in the closed interval 0 to 255.

- Each "stateset" must be defined in such a way that all its state elements belong to the first 255 state identifiers defined for the current body. E.g., if state-definition is "state s1, ..., s255, ..., sn;", then the stateset-definitions

    stateset A=[s187..s205];
    stateset B=[s1..s255];

    are both admissible, and the stateset definitions

    stateset C=[s1..s256];
    stateset D=[s250..sn];
    stateset E=[s255,s256]

    are all not accepted.

- The type "integer" cannot be used as an index type (e.g., the use of type denoter "array [integer] of T" is not allowed)².

Extensions

The language accepted by EDT tools is extended in three directions.

First, certain comments are processed and not, as usual, ignored. These are comments starting with a dollar sign ('$') as their first significant character, e.g.:

($text of a comment$) and {text of a comment}.

Such special comments are called qualifying comments. They may only appear in a precisely defined place in a specification. Appendix 1 contains a definition of these places together with a description of how the qualifying comments are processed.

² The restriction applies only to Estelle C-code Generator in that a specification which does not follow the restriction is accepted by Estelle Translator and is rejected in the generation phase.
The purpose of qualifying comments is to provide the possibility of inserting transparent directives into Estelle specifications. These directives are devoted to tools dealing with such Estelle specifications. It is also a way of annotating the Estelle specifications, i.e., to include some information which may be useful for some purposes in further processing but which is application specific and as such is not a part of an Estelle specification.

A second extension, make it possible to nest transitions using a transition "name" keyword. It is useful, in particular, to specify in a nondeterministic way the actions to be performed when the transition is enabled. For example, the following transition definition is accepted:

```
trans when ip1.m0
  name t1:
    begin output ip1.m1 end;
  name t2:
    begin output ip2.m2 end;
```

Third, it is allowed to use directives (e.g., #include <file>, #define) within an Estelle text in the same way as in the C language.

### 3. SYSTEM REQUIREMENTS AND INSTALLATION

#### Requirements
For real-life Estelle specifications EDT tools need about 4MB of internal memory and about 10MB of the user disk space for execution. About 4MB of disk space must be reserved for storing the EDT package itself. An additional space is needed for the X window (XEDT) interface.

#### Installation
Once the ESTELLE DEVELOPMENT TOOLSET (EDT) package has been installed under a Unix system the user has to modify his UNIX_shell environment:

- Add the ESTEL shell variable (and export it, if necessary) to define the absolute path to the directory where the ESTELLE DEVELOPMENT TOOLSET package is installed;
- Add the path name $ESTEL/bin in the PATH shell variable;
- Add the CPPPATH shell variable (and export it, if necessary) defining the absolute name of the CPP pre-processor. The absolute name of the CPP pre-processor is composed of the absolute path to the directory where the CPP pre-processor is installed followed by the CPP file name. The definition of the CPPPATH shell variable is only necessary if the absolute name of the CPP pre-processor in your environment is different from /usr/lib/cpp (which is the default value of the CPPPATH shell variable).

The user may also define three other shell variables, namely, LINES, LIB and EDITOR. The LINES shell variable is used within the Edb simulator/debugger for defining the number of lines to be displayed after which the control will be given back to the user (default value is 24) who has to type <RETURN> to continue displaying information. The LIB shell variable is used for defining the file names containing User Libraries to be linked with the C-code generated by the Estelle Compiler (Ec). The EDITOR shell variable is used to set the default text editor (vi or emacs).

The user has also to verify that he has the necessary environment to access to the Gnuplot and Ghostview tools (used by Edb simulator/debugger invoked to plot and view curves if it is NOT invoked through the XEDT graphic interface).
Additional shell variables should be defined when EDT tools are invoked through the XEDT graphic interface. Please consult XEDT User Reference Manual.

**Note:**
If you use a *csh*-like shell environment then the following text can be included in, for example, `.cshrc` file:

```bash
setenv ESTEL <absolute path to the dir. where EDT is installed>
setenv CPPPATH <absolute name of CPP pre-processor>
setenv EDITOR emacs
set path=$(ESTEL/bin $path)
```

If you use a *sh*-like shell environment then the following text can be included in the `.profile` file:

```bash
ESTEL=<absolute path to the dir. where EDT is installed>
export ESTEL
CPPPATH=<absolute name of CPP pre-processor>
export CPPPATH
EDITOR=emacs
export EDITOR
PATH=$ESTEL/bin:$PATH
export PATH
```

### 3.1 License environment

FLEXlm (Flexible License Manager) which is a network-wide floating licensing package protects EDT tools. It may restrict the number of simultaneous access to EDT tools and usage duration.

If EDT tools are protected by FLEXlm the following 4 files must be present in the directory `$ESTEL/bin`:

```
lmgrd
dedt
edt.dat
edtflexlm
```

The license manager daemon (*lmgrd*) handles the initial contact with the EDT/XEDT tools, passing the connection on to the *dedt* (vendor) daemon. It also starts and restarts the *dedt* daemon.

The *dedt* daemon is the license server. It keeps track of how many licenses are checked out, and who has them. If the *dedt* daemon terminates for any reason, all users lose their licenses (though this does not mean the tools suddenly stop running). Users normally regain their license automatically when *lmgrd* restarts the *dedt* daemon, though they may exit if the *dedt* daemon remains unavailable.

The file *edtflexlm* is a shall-script that runs the *lmgrd* daemon.

Licensing data is stored in a text file *edt.dat*, called the license-key file.
The *edt.dat* file looks like:

```
#----------------------------------------
SERVER moon 80a30323
VENDOR dedt
FEATURE GENERATOR dedt 4.2 permanent 20 0306F6228313 ck=54
FEATURE TRANSLATOR dedt 4.2 permanent 20 2E936EB76CDA ck=162
FEATURE EDB dedt 4.2 permanent 20 F904BD796C80 ck=184
FEATURE UG dedt 4.2 permanent 20 0EEE8D1BA47B ck=174
FEATURE EDOC dedt 4.2 permanent 20 D342A3B06C55 ck=82
FEATURE HLD dedt 4.2 permanent 20 38EDE29C749F ck=193
FEATURE XEDT dedt 3.1 permanent 20 33CE96B87359 ck=103

#NOTE: You can edit the hostname on the server line (1st arg).
#      The (optional) daemon-path on the VENDOR line (2nd arg).
# Most other changes will invalidate this license.
#----------------------------------------
```

In each FEATURE line there is a license expiration date. If the date is "permanent" then no time limit is set. If you invoke an EDT tool and a following error message is displayed:

```
EDB: feature has expired
```

it means that your license have expired.

In each FEATURE line, a number just after the license expiration date indicates an allowed number of simultaneous access to a specified tool (GENERATOR, TRANSLATOR, EDB, etc.). If the number of users who are simultaneously running this tool reach the allowed number, the following error message will be displayed:

```
EDB: no more license token available, please try later on
EDB: licensed number of users already reached
```

In each FEATURE line just after "dedt" the version of the tool is indicated. If the version of the invoked tool doesn't match this version a following error message will be displayed:

```
EDB: license file does not support this version
```

In these three above mentioned cases you have to obtain a new *edt.dat* file.

To install the license server you have to log-in on the host named after the keyword SERVER, and run `$ESTEL/bin/edtflexlm`.

If you invoke an EDT tool and a following error message is displayed:

```
EDB: cannot connect to license server (Connection refused)
```

it means that the licence server is not running and you have to log-in on the host named after the keyword SERVER, and run `$ESTEL/bin/edtflexlm.script`.
4. PRIMITIVE FUNCTIONS AND PROCEDURES IN EDT

Some functions and procedures may be declared within an Estelle specification as *primitives*. In this case only their headers are declared, followed by the key word *primitive*. Their bodies, which describe what these functions and procedures actually have to do, may be seen as implementation dependent and, as such, left for further considerations.

The problem arises when one decides to simulate such a specified system or has to generate a prototype implementation. In such cases the behaviour of primitive functions and primitive procedures declared within the Estelle specification have to be written in C and linked with the C-code generated during the compilation of the Estelle specification.

The C code generated by the Ec compiler is written in the, so-called, *traditional* -C. The same version of the language should therefore be used for writing in C the *primitive* functions and procedures whose headers were defined in Estelle. It must be remembered that several differences exist between this *traditional*-C and the standard ANSI-C version.

When writing the primitive function and procedures in *traditional* C, the following rules have to be respected in order to link them with the C-code generated by the Ec compiler:

• the identifiers used within the declarations of functions and procedures in C have to correspond to the names used within the Estelle specification but should be written using only **lower case** letters. For example

<table>
<thead>
<tr>
<th>in Estelle</th>
<th>in C (developed by the User)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE EXAMPLE: primitive</td>
<td>void example()</td>
</tr>
</tbody>
</table>

• the following correspondence between types in Estelle and in C should be respected:

<table>
<thead>
<tr>
<th>in Estelle</th>
<th>in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>REAL</td>
<td>double</td>
</tr>
<tr>
<td>CHAR</td>
<td>char</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>unsigned char</td>
</tr>
<tr>
<td>enumeration</td>
<td>enum</td>
</tr>
<tr>
<td>RECORD</td>
<td>struct</td>
</tr>
<tr>
<td>ARRAY OF TYPE</td>
<td>type[]</td>
</tr>
</tbody>
</table>

************************************************************************

3 The existence of several differences between them is due to stronger type checking within ANSI-C. For example in ANSI-C both the argument list and the return type of every function call are type-checked during compilation. In both versions, however, for historical reasons, it is not an error, that a function, for which the *void* type was not explicitly declared, do not return a value (only a warning will be issued, in general). Another difference exists within the function argument list declaration (not directly typed in *traditional* C). For example:

<table>
<thead>
<tr>
<th>'traditional'- C</th>
<th>ANSI-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>power(base, n)</td>
<td>power(int base, int n)</td>
</tr>
<tr>
<td>int base, n</td>
<td>[body]</td>
</tr>
</tbody>
</table>

© 1989, 1990, BULL S.A
© 1991 - 2000, INT
For example:

**in Estelle:**

```plaintext
TYPE
    WeekDay = (Sunday, Monday, Saturday);

PROCEDURE New_Agenda(Day: WeekDay); primitive;

FUNCTION SMOOTHING(Val: REAL) : REAL; primitive;
```

**in C developed by the User:**

```plaintext
/*begin of file: example.c */
enum weekday = {Sunday, Monday, Saturday};

void new_agenda(day)
    weekday day;
{
    /* appropriate C code here, which describes the behaviour of the new_agenda procedure*/
}

double smoothing(val)
    double val;
{
    /* appropriate C code here, which describe the behaviour of the smoothing function*/
}
/*end of file: example.c */
```

**Remark:** Note that the identifiers used within the type declarations in C may be different from those used in Estelle. For example, instead of the definitions mentioned above the following could have been used:

```plaintext
enum semaine = {dimanche, lundi, samedi};

void new_agenda(jour)
    semaine jour;

double smoothing(x)
    double x;
```

**Remark:** The file `example.c` has to be compiled (`cc -c example.c`) in order to generate the file `example.o`

The Estelle Compiler (Ec) and Estelle simulator/debugger (Edb) tools provide special facilities to deal with specifications in which some functions and procedures are declared as `primitive`.
Estelle-to-C Compiler (Ec)

The Estelle compiler is to be invoked with the M and L options (and without the d option\(^4\)):

\[
\text{ec <other options> -M -L<lib> <source_file_name>.stl}
\]

to generate the makefile <source_file_name>.mk.

The <lib> is the object file name (e.g., example) containing the body description of the primitive functions and procedures used within an Estelle specification.

The option L may be used several times (10 times is currently allowed), for example

\[
\text{ec <other options> -M -Lexample1.o -Lexample2.o <source_file_name>.stl}
\]

When several object files have to be linked, another possibility is to build an archive by using the UNIX "ar" command, and then use the L option followed by the name of the archive:

\[
\text{ec <other options> -M -Lexample.a <source_file_name>.stl}
\]

Another possibility is to invoke the Estelle compiler without L option and to modify the generated makefile <source_file_name>.mk. Within this makefile the line LIB has to be completed with the object file <lib>, where <lib> is the object file name (e.g., example.o) containing the body description of the primitive functions and procedures used within an Estelle specification. For example:

\[
\text{LIB = <existing files> example.o}
\]

where <existing files> contains all libraries automatically included.

Similarly, when many object files have to be linked the line LIB has to be completed as follows:

\[
\text{LIB = <existing files> example1.o example2.o}
\]

or

\[
\text{LIB= <existing files> example.a}
\]

Note: To avoid manual modifications of the makefile <source_file_name>.mk the LIB shall variable may be added to the user Unix shall-environment (and exported, if necessary) to define the User Library files to be additionally linked.

Estelle simulator/debugger (Edb)

Edb has to be invoked with the L option, i.e.:

\[
\text{edb -L<lib> <other options> <source_file_name>.stl}
\]

where <lib> is the object file name (e.g., example) containing the body description of the primitive functions and procedures used within an Estelle specification.

The option L may be used several times (10 times is currently allowed), for example

---

\(^4\)When the Estelle Compiler Ec is invoked with options M and d the generated makefile 'spec.mk' contains information on all the libraries necessary to generate the executable code for the Edb simulator/debugger.
When several object files have to be linked, another possibility is to build an archive by using the UNIX "ar" command, and then use the L option followed by the name of the archive:

```
edb -Lexample.a <other options> <source_file_name>.stl
```

### 4.1 Estelle 'primitives' delivered with the EDT package

The EEprim.h (in $ESTEL/libegc) contains standard input/output Estelle 'primitive' functions and procedures (e.g., WRITE_INT, WRITELN_INT, WRITE_REAL, READ_INT, READ_REAL etc.), which have behaviour similar to Pascal WRITE, WRITELN procedures and READ function with respect to a given type of parameters (integer, real, boolean, character).

The file.o (in $ESTEL/libegc/Libegc.a) contains the corresponding library which is automatically linked when the C-code generated is compiled and linked with the implementation or simulation motors. The contents of the EEprim.h file is given in the Appendix 2.

**Remark:** Note that all functions of the C library in your environment may be transcribed as Estelle primitives. For example to define Estelle primitives for initializing the seed of the random number generator and for generating the random numbers it is sufficient to transcribe the corresponding C functions in Estelle:

```
function srand (seed: INTEGER): INTEGER; primitive;

function rand: INTEGER; primitive;
```

The corresponding library will be automatically linked when the generated C-code is compiled and linked with the implementation or simulation motors.

### 5. ERROR PROCESSING IN EDT

When the Estelle-to-C compiler (Ec) is invoked with the -D option then the C code generated is augmented with some additional functions with which it has a run-time error detection capability. This means, for example, that when executing a division the corresponding C-code provides a means to check first if this is not an attempt to divide by zero.

If a run-time error occurs during the execution of a transition, the program "jumps" to the end of the transition, without executing the rest of commands, and it gives control to the user with the display of an appropriate error message. If a run-time warning situation is detected, the program continues its computation but an appropriate warning message is issued.

#### 5.1. Pascal run-time errors

The Pascal run-time errors that are detected are those usually recognised by the Pascal compilers. They deal with:

- a subrange violation in expressions; this concerns assignments, value parameter passing, array indexing, and set element exceeding set type;
- an access through "nil" pointer;
- a division by zero;
- a "case" index not defined.
In each case an error message will be issued when executing the code (simulation/debugging or prototype implementation).

5.2. Estelle run-time errors and warnings

Strictly speaking, all run-time errors of pure Estelle constructs are caused either by

- an uninitialized module variable used in an executed Estelle construct, or
- a run-time error in a Pascal expression used within a specific Estelle construct and evaluated when the construct is executed.

Thus run-time errors of the non-standard Pascal part of Estelle do not really differ from those of Pascal and are treated accordingly. The errors may occur during execution of the following Estelle constructs:

- delay-clause;
- provided-clause;
- when-clause;
- init-statement;
- detach-statement;
- connect-statement;
- disconnect-statement;
- release-statement;
- terminate-statement;
- output-statement;
- exist-expression;
- forone-statement;
- assignment between two module variables;
- equality comparison between two module variables.

Few examples suffice to illustrate these errors.

Let E1 and E2 denote expressions admissible for a given Estelle construct used below.

delay(E1,E2)

A run-time error may occur while a timer is to be activated with values of E1 and E2 and a Pascal run-time error occur in E1 or E2.

when ip[E1].m

A run-time Pascal error may occur in E1, e.g., E1 evaluates to a value that is outside of the array bounds.

release X

X may be not initialised, i.e., X does not currently reference a module instance.

attach X[E1].ip1 to ip2[E2]
A run-time error may occur because there is a Pascal error in E1 or in E2, but also if E1 evaluates to a value for which the module variable X[E1] is not initialised.

\[\text{forone } x: T \text{ suchthat E1 do stm1 otherwise stm2 provided E1 exist x: T suchthat E1}\]

In all the three above constructs a run-time error may occur in E1 (note that E1 is not necessarily a pure Pascal boolean expression since it may use a module variable and/or it may include an exist-expression inside it).

**Remark:** Obviously, a run time error may occur within the statements \(\text{stm1}\) or \(\text{stm2}\) within the above forone-statement but, in that case, the concerned construct is that which caused the error and not the whole forone-statement.

\[X := Y \text{ or } X = Y \quad (X \text{ and } Y \text{ are module variables})\]

A run-time error occurs in case of the assignment if Y is not initialised, and in case of the comparison, if one of the variables X or Y is not initialised (these run-time errors are not yet signalised)

The execution of a specific Estelle construct in certain situations is not considered, strictly speaking, i.e., according to the Estelle semantics, an error, but it may indicate that the behaviour does not correspond to the intentions of the specification designer. There are several "standard situations" of that sort recognised by the Estelle debugger, for which a warning is issued. These situations are listed below.

1/ **incorrect delay-bounds**

While a timer is to be activated for a delay-transition with the delay-clause "delay(E1,E2)", it occurs that the current value of the expression E2 is less than the current value of E1, or the latter is less than 0. Semantically, in such a situation the timer is not activated.

2/ **incorrect use of attach-statement**

An attach-statement, within an executing transition of a module instance, attempts to attach two interaction points while one (or both) of them is (are) already bound (attached or connected) by the same module.

3/ **incorrect use of detach-statement**

A detach-statement, within an executing transition of a module instance, attempts to detach an interaction point which is not currently attached.

4/ **incorrect use of connect-statement**
A connect-statement, within an executing transition of a module instance, attempts to connect two interaction points while one (or both) of them is (are) already bound (attached or connected) by the same module.

5/ **incorrect use of disconnect-statement**

A disconnect-statement "disconnect ip", where ip is an interaction-point-access, attempts to disconnect an interaction point which is not currently connected to another one, or a "disconnect X" statement, where X is a module variable, is used while there is no external and connected interaction point of the child module instance referenced by X ".

6/ **incorrect use of output-statement**

An output-statement sends an interaction through an interaction point that is not currently an end-point of a link to another interaction point.

**Remark:** Semantically, statements 2-6 are ignored, i.e., they have no effect. For the case of an output-statement it is sometimes said that the output is "lost".

**REFERENCES**


APPENDIX 1 - QUALIFYING COMMENTS

The language accepted by EDT is extended in that certain comments are processed and not, as usual, ignored. These are comments starting with a dollar sign ('$') as their first significant character, i.e., comments of the form

\[(*$text \ of \ a \ comment*)\]

or

\[{$text \ of \ a \ comment}.$\]

Such special comments are called qualifying comments.

The purpose of qualifying comments is to provide the possibility of inserting transparent directives (for analysis and/or for generation) in Estelle specifications.

A qualifying comment may only appear in a precisely defined place of a specification. The rules given below must be obeyed by the user when inserting qualifying comments:

- Each occurrence of a comment starting with a dollar sign ('$') as its first significant character shall be scanned as occurrence of a qualifying comment;

- Qualifying comments may only occur within a list of qualifying comments;

- A qualifying comment list is a sequence of one or more qualifying comments, separated by no other text than regular comments, i.e.: blanks, newlines, tabs and regular Pascal comments;

- Occurrence of a qualifying comment list is always optional, i.e., no one is forced to put a qualifying comment list everywhere it is allowed;

- Occurrence of a qualifying comment list shall be allowed only in the three following cases:

  1+ Immediately before an identifier, if the occurrence of this identifier is its defining point;

  2+ Immediately before an imperative statement, even before an empty statement;

  3+ Immediately before an expression ("expressions" should not be confused with "terms" and "factors" !).

- An occurrence of a qualifying comment list outside the three listed above is an error.

The object of an Estelle specification ("identifier", "expression" or "imperative statement") with which a qualifying comment list is associated, is called qualified object.

Processing of qualifying comments by the translator.

The Estelle Translator does not concerne itself with the text of a qualifying comment. It only copies the text of each qualifying comment - once stripped of its delimiters (i.e. : "('*$*' '*')" or "'{*' '*}.'" - into
the Intermediate Form representation of the analysed specification, with appropriate links to the rest of the objects described in this representation.

The qualifying comments of a qualifying comment list, are placed into Intermediate Form in such a way that the initial sequencing of these comments is preserved.

**Processing of qualifying comments by the C code generator.**

The C code generator treats qualifying comments whose text in the IF-representation starts with: C$, R$, W$ provided the options -q, -R or -W have been used, respectively.

With the -q option the remaining text after C$ in IF-representation is supposed to be a piece of C-code text and it is inserted into the generated code. However, no C syntax checking is provided by the Estelle compiler for these inserted texts. For instance, from the original qualifying comment

\[
(*$C$ printf (".-")); *
\]

the part

\[
C$printf (".-");
\]

is copied into IF-representation, and the part

\[
printf (".-");
\]

is treated as a piece of C and it is inserted into the generated code.

With the -W option the remaining text after W$ in the IF-representation is supposed to be a piece of C-code declaration and it is inserted into the generated code. Note that no C syntax checking is provided by the Estelle compiler for these inserted texts. The qualifying comments of this sort have to be used to complete an Estelle type declared as "..." and to complete an Estelle constant declared as "any <TYPE>". The -W option have to be used with the -u option.

For instance, when such qualifying comments are used as follows in an Estelle specification (also see example given for the use of -R option):

```c

 type (*$W$ char * *) data_type = ... ;
 const (*$W$ 0 *) C = any INTEGER;

 then the following text is generated

 typedef char * r<N>_data_type;
 #define r<N+1>_C 0

 With the -R option the text R$ in IF-representation is a directive for the C code generator which prevents renaming the identifier which follows the qualifying comment (*$RS*$).

```

© 1989, 1990, BULL S.A
© 1991 - 2000, INT
Last change: August 21, 2000
For instance, when the qualifying comments (*$R$*) are used as follows in an Estelle specification (also see example given for the use of -W option):

\[
\text{type } (*$R$*) (*$W$ char * *) \text{ data_type} = \ldots ;
\]

\[
\text{const } (*$R$*) (*$W$ 0 *) \text{ C = any INTEGER};
\]

\[
(*$R$*) (*$W$ (datatype) 0 *) \text{ NULL_DATA} = \text{ any data_type};
\]

then the following text is generated

\[
\text{typedef char * data_type};
\]

#define C 0

#define NULL_DATA (datatype) 0
APPENDIX 2 - STANDARD I/O ESTELLE PRIMITIVES

(**********************************************************************)
(* (1a) Use the declarations of functions and procedures as defined in this file *)
(* directly in the declaration part of your ESTELLE specification_body, *)
(* or *)
(* (1b) Use #include "EEprim.h" in the declaration part of the *)
(* ESTELLE specification_body to include this file, and use the *)
(* option -I<$ESTEL/libgec> when the Estelle compiler (Ec) *)
(* or simulator/debugger (Edb) is invoked *)
(**********************************************************************)

****READ, WRITE and WRITELN for INTEGER-TYPE VARIABLE*****
(* read and write of an integer on the standard i/o *)
(* . READ_INT *)
(* . WRITE_INT ( without a jump to a new line ) *)
(* . WRITELN_INT ( with a jump to a new line ) *)
(* Usage: *)
(* I := READ_INT *)
(* WRITE_INT( I ) *)
(* WRITELN_INT( I ) *)

function READ_INT : INTEGER ; primitive ;
procedure WRITELN_INT( I : INTEGER ) ; primitive ;
procedure WRITE_INT( I : INTEGER ) ; primitive ;

****READ, WRITE and WRITELN for a CHARACTER-TYPE VARIABLE**
(* read and write of a character on the standard i/o *)
(* . READ_CHAR *)
(* . WRITE_CHAR ( without a jump to a new line ) *)
(* . WRITELN_CHAR ( with a jump to a new line ) *)
(* Usage: *)
(* C := READ_CHAR *)
(* WRITE_CHAR( C ) *)
(* WRITELN_CHAR( C ) *)

function READ_CHAR : CHAR ; primitive ;
procedure WRITELN_CHAR( C : CHAR ) ; primitive ;
procedure WRITE_CHAR( C : CHAR ) ; primitive ;
(******READ, WRITE and WRITELN for a REAL-TYPE VARIABLE******)
(*
(* read and write of a real on the standard i/o
(*   . READ_REAL
(*   . WRITE_REAL ( without a jump to a new line )
(*   . WRITELN_REAL ( with a jump to a new line )
(*
(* Usage: (*
(* R := READ_REAL (*)
(* WRITE_REAL( R ) (*)
(* WRITELN_REAL( R ) *)
(*
(***************************************************************)

function  READ_REAL   : REAL ; primitive ;
procedure WRITELN_REAL( R : REAL ) ; primitive ;
procedure WRITE_REAL( R : REAL ) ; primitive ;

(****READ, WRITE and WRITELN for a BOOLEAN-TYPE VARIABLE***)
(*
(* read and write of a boolean on the standard i/o
(*   . READ_BOOL
(*   . WRITE_BOOL ( without a jump to a new line )
(*   . WRITELN_BOOL ( with a jump to a new line )
(*
(* Usage: (*
(* BOOL := READ_BOOL (*)
(* WRITE_BOOL ( BOOL ) (*)
(* WRITELN_BOOL ( BOOL ) *)
(*
(***************************************************************)

function  READ_BOOL   : BOOLEAN ; primitive ;
procedure WRITELN_BOOL( BOOL : BOOLEAN ) ; primitive ;
procedure WRITE_BOOL( BOOL : BOOLEAN ) ; primitive ;