ESTELLE - TO - C COMPILER

Ec

USER REFERENCE MANUAL
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  
Email: stan@int-evry.fr

*Note:* The following persons participated in the Ec development:

P. Ansart, V. Chari, T. Claes, T. Gilot, E. Lallet, S. Mariel, J.F. Martin; E. Mattera, E. Mariani, J.L. Richard, J.D. Ruff.

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.

The information in this document is subject to change without notice and should not be construed as a commitment by INT.  
INT takes no responsibility for any errors or omissions that may exist in this document or for damages resulting from the use of the information herein.

Copyright © 1989-90 Bull S.A  
Copyright © 1991-2000 INT  
All rights reserved
TABLE OF CONTENTS

1. INTRODUCTION.................................................................................................................. Erreur! Signet non défini.
2. INPUT LANGUAGE.............................................................................................................6
   2.1 Restrictions.................................................................................................................... 6
   2.2 Extensions.................................................................................................................... 7
3. COMPILER INVOCATION .................................................................................................8
   3.1 Translator options........................................................................................................ 9
   3.2 Generator options.......................................................................................................10
4. COMPILER OUTPUTS.........................................................................................................12
   4.1 Compiler warnings and errors................................................................................... 12
   4.2 Cross-reference listing.............................................................................................. 12
   4.3 Intermediate Form...................................................................................................... 12
   4.4 Organisation of the generated C code.......................................................................13
5. IMPLEMENTATION OR SIMULATION KERNEL INTERFACE......................................15
   5.1 Predefined types for implementation or simulation................................................. 15
   5.2 Macros to be offered by Estelle Implementation or Simulation Kernel...................... 15
   5.3 Primitives of runtime libraries.................................................................................. 17
6. AUTOMATIC PROTOTYPE IMPLEMENTATIONS.............................................................20
REFERENCES......................................................................................................................26
this page is intentionally left empty
1. INTRODUCTION

This document describes an Estelle-to-C compiler, which is an extended version of the compiler developed by BULL S.A. and MARBEN [15] based on a prototype compiler designed within the ESPRIT/SEDOS\(^1\) project [13,14].

Estelle compiler is a part of the ESTELLE DEVELOPMENT TOOLSET (EDT) package which includes the Estelle Simulator/Debugger – Edb and many other tools.

Since the EDT version 3.2 the restriction (w.r.t the C-code generator) concerning the simultaneous occurrence within a transition declaration of an any-clause with a priority-clause and/or a delay-clause no longer applies.

The Ec compiler integrated in the EDT since its version 4.1 has less restrictions w.r.t the input language and in particular detects (error or warning messages – see the option \(\text{-N}\)) whether the functions are pure. It allows also to compile Estelle text stored in files whose names has unlimited length.

Since the EDT version 4.2, the generated C code is optimised w.r.t. the usage of ‘any-clauses’.

The Estelle Compiler translates an Estelle specification into traditional [9] C language source code. The compiler consists of the following two tools:

- a **translator** (Estelle Translator)\(^2\) which given an Estelle specification returns the specification representation in so called Intermediate Form (IF) (see [7], [16]), augmented with information resulting from a complete static analysis (lexical, syntactical and semantical) of the specification.

- a **C code generator** (Estelle Generator)\(^3\) which given the Intermediate Form representation of an Estelle specification returns the C-code of the specification.

By default, the C-code generator is automatically invoked once the translator has terminated generating the IF-representation to produce C-code directly from the Estelle source text. The compiler may be also used separately as the above translator or the above C-code generator provided an IF-representation is already available (sees options in Section 3).

The purpose of introducing the Intermediate Form representation is to enable various application-oriented tools (e.g. simulator/debugger, browser, splitter) or different code generator to work with the common, suitably organised, and already verified representations of Estelle specifications. In particular besides the traditional [9]C-code actually generated, the C++ or Java can be considered etc.

---

\(^1\) European Strategic Programme for Research and Development in Information Technologies / Software Environment for the Design of Open Systems.

\(^2\) The first versions of the translator and its Intermediate Form were written in PASCAL, within the project ESPRIT/SEDOS, at Agence de l’Informatique [1], [2]. The second version was developed in PASCAL at BULL S.A. The third version (written in C) was developed mainly by MARBEN. The translator was designed using the automatic tool SYNTAX developed in INRIA [11].

\(^3\) The first version of the generator was written in PASCAL within the project ESPRIT/SEDOS, at Bull S.A [14]. It is specified in [3] and based on the study [4]. The second and third version was written in C at BULL S.A.
The compiler has been tested with respect to purely Estelle and Pascal constructs. The BSI (British Standard Institute) validation sequences for Pascal compilers have been used to test the Pascal part of Estelle.

The output files of the Estelle Compiler (see Section 4) mainly contain a portable, operating system independent, C representation of components of a source Estelle specification.

To get an executable code of an Estelle specification on a given host system, some additional pieces of the C-code have to be linked to the C-code generated by the compiler. The main part of this additional code form an Estelle Implementation Kernel (also called Implementation motor). The specification of this kernel [8] includes the definitions of types, macros and primitives which are used, but not provided, within the C-code generated by the Estelle Compiler. The Implementation Kernel for the UNIX operating system has been designed by MARBEN, was extended by INT and is a part of the EDT package (the library $ESTEL/estmote/libecm.a).

The Estelle specification of a system may contain several subsystems (modules attributed systemprocess or systemactivity) supposed to run in parallel asynchronous way. Such a system may be implemented as a single process running under a UNIX operating system or as a set of processes distributed over a network (see Section 6).

The document is organised as follows. Section 2 describes the input language for the Ec compiler. Section 3 gives information about variants of the compiler invocation under a UNIX system. Section 4 is devoted to a brief description of compiler outputs under different options. Section 5 defines the interface between the Estelle Compiler and Estelle Implementation and simulation Kernels [8]. Section 6 gives some procedures and hints to implement Estelle specified system in a centralised or distributed way.

2. INPUT LANGUAGE

Estelle as described in ISO 9074 [5] is the input language of the compiler. Note that for technical reasons the Estelle grammar of the compiler (the grammar is LALR(1) type ) differs from that of ISO 9074 document. The language is the subject of some minor restrictions and extensions that are listed below. They are mainly due to machine-dependent limitations, but some of them were introduced to ensure a coherent processing of the Intermediate Form by generators independently from their orientations (implementation, simulation, etc.) and/or to direct this processing by means of so called qualifying comments.

2.1 Restrictions

The compiler accepts at its input an Estelle text with the following restrictions:

- 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 analyser.

- 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 s_1,...,s_{255},...,s_n.", then the stateset-definitions

\[
\text{stateset } A = [s_{187}, s_{203}]; \quad \text{stateset } B = [s_{1}, s_{255}];
\]

are both admissible, and the stateset definitions

\[
\text{stateset } C = [s_{1}, s_{256}]; \\
\text{stateset } D = [s_{250}, s_{1}]; \\
\text{stateset } E = [s_{255}, s_{250}]
\]

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)\(^4\).

2.2 Extensions

The language accepted by the compiler 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.:

\[
(*$text of a comment*) \text{ and } \{text of a comment}\]

Such special comments are called qualifying comments .

A qualifying comment may only appear in precisely defined places of a specification. Appendix 1 in the document 'EDT - General Information' contains a definition of these places together with a description of how the qualifying comments are processed by the compiler (see also generator options "-q", "-R" and "-W" in Section 3.3).

A second extension, make it possible to nest transitions using a transition "name" keyword. Please consult Section 3 of the document 'EDT - General Information' for more information.

A third extension, make it possible to use within an Estelle text the directives #include <file>, #define, in the same way as in the C language.

2.3 Deviancies

- The following scoping rule of the Pascal standard ([10]) is not implemented:

\[
\text{"Within the scope of a defining-point of an identifier or label, there are no applied occurrences of an identifier or label that cannot be distinguished from it and have a defining-point for a region enclosing that scope".}
\]

\(^4\) 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.
In other words, the Estelle compiler would not recognise as erroneous a declaration of an object that has been previously used in the same scope. For example,

```pascal
type t0=integer;
procedure p;
type
t1=t0;
t0=char;
begin ...
end;
```

is formally incorrect in Pascal (Estelle), because symbol "t0" is defined within the scope where it was previously used, but would not be recognised as such by the Estelle compiler.

- The compiler does not check the congruity of the following parameter lists (which are not congruous):

```pascal
i, j: integer  ...

and thus they are not distinguished by the compiler.
```

The above deviances have no consequences on Estelle use and semantics.

### 3. COMPILER INVOCATION

The Estelle Compiler, once installed under a UNIX system, is directly invoked from the UNIX-shell environment by the single command described below:

```
ec [phase options] [translator options] [generator options] file
```

The "phase options" are: -t and -g. These two options indicate, respectively, that the compiler is to be only used as Estelle Translator with options specified by "translator options", or it is to be only used as Estelle Generator with options specified by "generator options".

By default (i.e., with no "phase option" specified) the ec command invokes the whole Estelle Compiler, that is, it the C-code generator is automatically invoked after the translator, if the later succeeded. In this "default" case both "translator options" and "generator options" may appear.

When the whole compiler (default option) or the translator (-t option) is invoked, "file" is the name of a file that contains an Estelle source specification. The name must have the following form:

```pascal
<source_file_name>.stl
```

When the generator (-g option) is invoked, "file" is the name of a file which contains the intermediate form representation of an Estelle specification (the output of translator). For that case, the name must be

```pascal
<source_file_name>.if
```

---

3 Please note that the "spec" identifier is NO MORE limited to 10 characters. The "spec." may correspond to the absolute path name.
An invocation of the Estelle Translator (ec -t) with the file '<source_file_name>.stl' and without any of the translation options performs:

 - the complete lexical and syntactical analysis, and
 - the complete static semantic analysis.

The output of the translation phase, if no error was detected consists of the Intermediate Form (IF) representation (Section 4.3) of the Estelle source specification. The IF-representation is contained in the file:

<source_file_name>.if

For an erroneous specification, no Intermediate Form is produced. If qualifying comments appear in the source Estelle specification, then they are stripped from their delimiters and copied into the Intermediate Form (for details see Appendix 1 within the document 'General Information').

If the Estelle Generator is directly invoked (ec -g) with the file

<source_file_name>.if

or if it is automatically invoked by the compiler after the translation phase, then it produces a C-coded representation (if any, see option "-u") of the Estelle source specification. This representation is contained in various output files that may differ according to chosen options (see Section 4.4).

During the translation phase and generation phase warning messages of these phases are displayed on the "standard output", e.g., on the screen.

It should be noted that there is no prescribed order in writing options. The user is informed by the system if an incoherent combination of options were used. A detailed description of the translator and generator options is given below.

### 3.1 Translator options

- **-i**
  With this option the translator does not produce the IF-representation of the source specification even if there is no error detected. Only syntactic and (static) semantic analysis is performed. When this option is used with the -l option it prevents the generation of the <source_file_name>.cpp.

- **-e**
  With this option error and warning messages are not displayed during translation (the option has no influence on the production of the translation listing).

- **-m**
  With this option the translation listing (the file <source_file_name>.t.l ) is produced with all warning and/or error messages issued during the syntactic and semantic analysis. A complete error recovery process is integrated during this analysis, which means that no error can abort the translation process.

- **-c**
  With this option the translation listing is augmented with a cross-reference listing (see Section 4.2).
-N

With this option the translator produce warnings instead of errors when non pure functions were used in the translated specification.

-I<dir>

With this option the user may indicate the path to the directory where the files to be included in the Estelle specification <source_file_name>.stl are stored. This files are specified in <source_file_name>.stl by means of the #include "file_name" directives inserted in it. When the option -I is used the CPP pre-processor is first invoked and the file <source_file_name>.cpp is generated (if the -i option is used this file is not generated) which contain the Estelle text after the inclusion of given files. Note that when the option -I is used the # characters can be used in the first column of the Estelle text only within the "include" context. The -I option may be used several times (it is currently limited to 10 times).

3.2 Generator options

-I

With the -l option, the correspondence is given between a line in the source Estelle specification and the blocks of C-code generated for it. This correspondence is in the form of a C comment in the generated code. Without this option no such correspondence is indicated.

-p

This option makes it possible to translate a pure Pascal program (the subset of ISO Pascal used in Estelle, see Annex C of ISO 9074) into the corresponding C program. In order to do that it is necessary:

- to replace the Pascal keyword "program" by the keyword "specification" in the source Pascal program,

- to add the keyword "initialize" before the main "begin" and to add "end;" before the last "end" in the source Pascal program.

If -p option is given, the generator outputs the unique C file '<source_file_name>.c'. The C compiler can be directly invoked by the command

cc -o <source_file_name> set.o pasc.o ptoc.o -I. <source_file_name>.c

-q

The -q option treats "qualifying comments" whose text, in IF-representation, begins with CS$ . The remaining text after CS in IF-representation is supposed to be a piece of C-code text and it is inserted into the generated C-code. However, no C syntax checking is provided by the compiler for these inserted texts. Without -q option such qualifying comments are recopied as C-comments (i.e., C-comment delimiters "/*" and "*/" are added to the copied text). All other qualifying comments (i.e., those which do not begin with "CS") are always recopied as C-comments provided the -W or -R option is not used (see below). Please consult Appendix 1 of the General Information part of this document for further details.

-W

The option -W treats "qualifying comments" whose text, in IF-representation, begins with WS$. The remaining text after WS in 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 compiler for these inserted texts. Without -W option such qualifying comments are recopied as C-comments (i.e.,

\[\text{During the translation phase the "qualifying comments" are stripped from their original delimiters "(*$" and ")" or "/{$ and \}" and copied into the IF-representation.}\]
C-comment delimiters "/*" and "*/" are added to the copied text). All other qualifying comments (i.e., those which do not begin with "WS") are recopied as C-comments provided the -q or -R option is not used (see below). The -W option is to be used always with the -u option (see below). Please consult Appendix 1 of the General Information part of this document for further details.

The qualifying comments of this sort are used to complete an Estelle type declared as "..." and to complete an Estelle constant declared as "any <TYPE>".

-\textbf{R} \quad The option -R treats "qualifying comments" whose text (in IF-representation) is RS#. Without -R option such qualifying comments are recopied as C-comments (i.e., C-comment delimiters "/*" and "*/" are added to the copied text). All other qualifying comments (i.e., those which do not begin with "RS") are recopied as C-comments provided the -q or -W option is not used (see above). Please consult Appendix 1 of the General Information part of this document for further details.

The qualifying comment of this sort is used in the Estelle declaration part to prevent renaming the identifier which follows such qualifying comment.

-\textbf{u} \quad With this option an incomplete C code is generated for an incomplete Estelle specification i.e., a specification containing constants declared "any <TYPE>", and/or types declared "...", and/or module bodies defined "external". For each "undefined" object of the above sort a "warning comment" is inserted into the generated code, i.e., the warning message occurs between C comment delimiters. The same warning comments are displayed on the "standard output".

Without the -u option no code is generated for an incomplete Estelle specification and an error is reported.

-\textbf{r} \quad With the -r option all identifiers (excepts names of procedures and functions declared as "primitive") of the compiled Estelle specification are renamed. Otherwise some of them remain the same in the generated code. It is appropriate to use this option if the source Estelle specification uses C keywords as identifiers or if some of the identifiers are too long for the C compiler.

-\textbf{w} \quad With this option an additional file

\begin{verbatim}
<source_file_name>.g.l
\end{verbatim}

is generated. The file contains all warnings and errors of the generation phase.

-\textbf{O} \quad This option allows the generation of a C code which checks, in runtime, for overflows of the "flattening stacks". For performance reasons this check is not included in the standard generation. The option is to be used when execution of the code generated without this option led to a bad memory access.

When during the execution of the code generated with -o option an overflow is detected, the runtime message "_STACK overflow" or "_DISPLAY overflow" is displayed (on the "standard output"). In such a case, the numeric constants _TOPSTACK or _TOPDISPLAY in the include file Mmacropile.h, must be increased and the code has to be recompiled.

-\textbf{d} \quad This option has to be present when the generated code is to be used by the Estelle simulator/debugger (Edb).
-D This option has to be present when one wants "run-time Pascal errors detection capability". It can be used either by an Estelle implementation motor or by a simulator/debugger (Edb).

-M With -M option a MAKEFILE named <source_file_name>.mk is generated. This makefile permits the automatic compilation of the generated C code (use the command make -f <source_file_name>.mk). If the option -d has been simultaneously used the generated C code is linked with the simulation motor (the generated file is <source_file_name>.edb which is the executable code of the Edb simulator/debugger). If the option -d has not been simultaneously used the generated C code is linked with the implementation motor (the generated file is <source_file_name> which is the executable code of a prototype implementation of the specification.

-L<library> With this option the user may indicate the name of the library in which the bodies of procedures and functions, declared within the Estelle specification as "primitive", are stored. The -L option may be used several times (it is currently limited to 10 times).

4. COMPILER OUTPUTS

4.1 Compiler warnings and errors
The Estelle Compiler warning messages inform the user that a "not-quite-normal" situation occurred but either it is not formally forbidden (like, e.g., duplicated declarations) or the correction was "obvious" for the compiler (e.g., the substitution of "="., erroneously used within an assignment, by ":=".). That is why warnings do not prevent the translator from producing the IF-representation of the analysed Estelle specification. They do not, in general, prevent the code generation. The only exception is when the compiler (generator) processes an incomplete Estelle specification without "-u" option (see Section 3.2). All warning messages in the compiler listing begin with "**** Warning:".

All compiler messages, which are not warnings, are error messages. An erroneous situation does not stop the analysis but leads to abort the compilation process before the Intermediate Form is produced. All error messages in the compiler listing begin with "**** Error:"

The warning and error messages of the translation phase are stored (if the option -m was used) in the file <source_file_name>.t.l and warning and error messages of the generation phase (if the option -w was used) - in the file <source_file_name>.g.l

4.2 Cross-reference listing
When the translation option -c is used the translation listing in the file <source_file_name>.t.l is augmented by a cross-reference listing which synthesises information derived from the translation analysis.

4.3 Intermediate Form
The Intermediate Form (IF) is an abstract representation of an Estelle specification [7]. It contains all information contained in the source Estelle text organised within structures that facilitate the use of different software tools, such as the C-code generator and the Edb simulator/debugger, for its manipulation.
The Intermediate Form (IF) file `<source_file_name>.if` is produced by the Estelle Translator once no errors was detected during the static check of the specification. It separates the static analysis phase from the generation phase. Currently, the C-code is generated from IF but other codes could be also generated without the necessity to rewrite the translator. For example, the ML (abstract tree) generator has been also developed for the ESTIM simulator [18].

All information in the IF-file is available for the user through a programming interface. The detailed description of the Intermediate Form organisation and of the access to it is provided by the document EDT- Intermediate Form: Utilisation principles.

### 4.4 Organisation of the generated C code

#### 4.4.1 Generated files

The generator produces, in the user's current directory, five types of files:

- A file named `<source_file_name>.d.c` which contains additional information needed by the Estelle simulator/debugger. This file is created only if the options `-d` is given by the user of the generator.

- Two files named `<source_file_name>.h` and `<source_file_name>.c` which contain references for the implementation and simulation motors, respectively.

- Files which contain the translation of the source Estelle specification into C. There are two files named `<BODY_NAME><num>.h` and `<BODY_NAME><num>.c` for each module `BODY_NAME` appearing in the Estelle source specification, and two files `<SPECIFICATION_NAME><num>.h` and `<SPECIFICATION_NAME><num>.c` for the specification body itself.

- A file named `<source_file_name>.mk`, which contains a MAKEFILE permitting the automatic generation of the executable code (use the command `make -f <source_file_name>.mk`). This file is created only if the options `-M` is used.

#### 4.4.2 Generated files contents

* The `<source_file_name>.d.c` file

This file comprises the "include" of the files described in section 4.4.3, the include of all generated files `<BODY_NAME><num>.h`, the declaration and the initialisation function `_initoffset` of an array `_offset[]` which gives the offset, in module contexts, of the Estelle elements declared in these contexts. This array is indexed by the entry numbers of the elements (see "Intermediate Form Reference Document" [7]).

* The `<source_file_name>.h` file

This file includes the constant definitions of the unique numbers associated with: each interaction type, module header type and module body declared in the source Estelle specification.

---

*From the version 4.2 there is NO MORE restriction consisting in taking as significant only the first 8 characters of `<BODY_NAME>` and of `<SPECIFICATION_NAME>`.*
* The `<source_file_name>.c` file

This file comprises the "include" of the files described in section 4.4.3, the include of the `<source_file_name>.h` file and either:

- with option debugger (-d): the initialisation function (_initdebug) of the array (_Tbodydesc[]) which contains the debugger descriptors of module bodies declared in the Estelle source, or

- without option debugger (-d): the declaration and static initialisation of the array of module header descriptors (_hdesc[]), the declaration and the initialisation function (_implemspec) of the array of module body implementation descriptors (_bdesc[]), and the main function of the implementation of the Estelle specification.

* The `<BODY_NAME><num>.h` and `<SPECIFICATION_NAME><num>.h` files

These files include:

- declaration of constants (which corresponds, in Estelle, to the constant-definition-part of the module body);

- declaration of types (which corresponds, in Estelle, to the type-definition part of the module body);

- declaration of interaction parameters (which corresponds, in Estelle, to the channel-heading, interaction-group and interaction-definition of the module body);

- declaration corresponding, in Estelle, to the definitions of the module headers appearing within the module body;

- declaration of the module context type.

* The `<BODY_NAME><num>.c` and `<SPECIFICATION_NAME><num>.c` files

These files include:

- the "include" of the files described in section 4.4.3, the include of the `<source_file_name>.h` file and the include of all generated files `<XXX>.h` associated with all module bodies `<XXX>` embedding the body identified by `<BODY_NAME>`;

- declaration of states defined in the module body;

- declaration of local functions (which corresponds, in Estelle, to the procedure-and-function-declaration-part of the module body);

- representation of the initialisation-part of the module body;

- representation of the transition-declaration-part of the module body.
4.4.3 Include files and libraries supplied by the generator

There is a set of routines to provide emulation of all PASCAL native operations which are not in the C language (set operations, ordinal operations and arithmetic function). The above routines are stored in object files pasc.o, set.o and ptoc.o contained in $ESTEL/libegc/Libegc.a.

The definitions which are required to use these routines are stored in include-files named "Mfunctext.h", "Mset.proc.h", "Mmacropile.h" and "Mcopy.h".

The definition (predefined constants and types, and primitives) of the interface with implementation or simulation motors is coded in the include file "estel.h".

5. IMPLEMENTATION OR SIMULATION KERNEL INTERFACE

This section provides a description of the interface with the Estelle implementation (or simulation) motor. Such motor, designed for a dedicated operating system, must be used to obtain an executable prototype implementation of an Estelle specification or an executable code for simulation.

Predefined types, macros and primitives of the runtime libraries which are described in the following subsections, are used in the generated C code. They are, however, provided by the implementation (simulation) motor, because they are dependent on the system on which Estelle is to be implemented.

NOTE: The code implementing this kernel under UNIX has been realised by MARBEN and extended by INT.

5.1 Predefined types for implementation or simulation

Definition of the following types must be written into the include file "estel.h":

- _MODVAR : type of module-variables,
- _ctxctrl : type of the structures used for storing control data and hierarchy data associated with a module,
- _intctrl : type of the structures used to control the interactions,
- _TYIPDT : type of interaction point descriptor,
- _ALLCTRL : type of control structures used to translate an all-statement that uses a module variable as its selector variable
- _DELAYTYPE : type of structures used to control an initiated delay operation (_setdelay()).

5.2 Macros to be offered by Estelle Implementation or Simulation Kernel

5.2.1 _mkctx:

synopsis: _mkctx (ctxp, mvarp, ctx_type, head_key, body_key)
description: This macro allows allocation of a context for a module instance of a given body type. ctxp points to the context of the calling module instance. mvarp is the address of the module variable where the result of this operation is memorised. ctx_type is the name of the typedef generated to define the type of the context associated with the child module to be instanciated. head_key is the unique number associated with module header type. body_key is the unique number associated with the body.

5.2.2 __runinit:
synopsis: __runinit (_modv, _body_key);

description: __runinit initialises the child module instance by executing one fireable transition among those described in the child initialisation-part. Parameters are the module variable (_modv) referencing the child module instance that must execute the one of its initialize transitions, and the unique number body_key associated with the body.

5.2.3 __mkinter:
synopsis: __mkinter (ctxp, inter_type, inter_key)

description: This macro permits the allocation of an interaction buffer for a given interaction type to store its parameters. ctxp points to the context of the calling module instance. inter_type is the name of the typedef generated to describe the interaction. inter_key is the unique number associated with the interaction type.

5.2.4 __outarg:
synopsis: __outarg (ctxp, inter_type, par_form)

description: This macro assigns the effective value of one interaction parameter; this interaction is previously allocated by __mkinter. ctxp points to the context of the calling module instance. inter_type is the name of the typedef generated to describe the interaction. par_form is the identifier of the formal interaction parameter in the interaction type.

5.2.5 __expvar, __expvarv:
synopsis: __expvar (mvar, header_type, var_id)
__expvarv (mvar, header_type, var_id)

description: These macros permit access to an exported variable or a parameter of a child module instance. If exported data is accessed without modification the __expvarv macro is used, otherwise it is __expvar.

- mvar is the module variable referencing the child module instance.
- header_type is the name of the structure generated to describe the module header.
- var_id is the identifier of the exported variable or parameter accessed.

5.2.6 __childipe:
synopsis: __childipe (mvar, header_type, ipe_id)

description: This macro permits access to an external interaction point of a child module instance. mvar is the module variable referencing the child module instance. header_type is the name of the structure generated to describe the module header. ipe_id is the identifier of the generated object for the external interaction point (type of this object is defined by __TYIPDT).
5.2.7 \texttt{modctx}:

\begin{verbatim}
synopsis: \texttt{modctx} (mvarp)
description: This macro gives the pointer to the context of a child module instance referenced by a given module variable (the type of this variable is \texttt{_MODVAR}).
\end{verbatim}

- mvarp is the address of the module variable referencing the child module instance.

5.3 Primitives of runtime libraries

5.3.1 \texttt{output}:

\begin{verbatim}
synopsis: \texttt{void \_output} (_adctx, \_ref_ip)

\_MODCTXYP \_adctx;
\_TYIPDT *\_ref_ip;

description: This function appends an interaction to the queue associated with the interaction point, which is linked to the interaction point whose reference (\_ref_ip) is passed as the input parameter. The second input parameter is a pointer to the context (_adctx) associated with the module instance executing the call.

\textbf{Note}: The set of interaction parameters is not provided by this call.
\end{verbatim}

5.3.2 \texttt{release}:

\begin{verbatim}
synopsis: \texttt{void \_release} (_p_ctxa, _p_mvar)

\_MODCTXYP \_p_ctxa;
\_MODVAR *\_p_mvar;

description: This function executes the release statement of Estelle language. The input parameters are the calling module context location (_p_ctxa), and the address of the module variable referencing the child module instance to be released (_p_mvar).
\end{verbatim}

5.3.3 \texttt{attach}:

\begin{verbatim}
synopsis: \texttt{void \_attach} (_p_ctxa, \_ipa, _p_ctxc, \_ipc)

\_MODCTXYP \_p_ctxa; \_TYIPDT *\_ipa;
\_MODCTXYP \_p_ctxc; \_TYIPDT *\_ipc;

description: This function executes the attach statement of Estelle language. The input parameters are the address of the calling module context (_p_ctxa), whose external interaction point (referenced by \_ipa), is to be attached to the external interaction point (referenced by \_p_ctxc) of the child module instance whose context address is \_ipc.
\end{verbatim}

5.3.4 \texttt{connect}:

\begin{verbatim}
synopsis: \texttt{void \_connect} (_p_ctx1, \_ip1, _p_ctx2, \_ip2)

\_MODCTXYP \_p_ctx1;
\_TYIPDT *\_ip1;
\_MODCTXYP \_p_ctx2;
\_TYIPDT *\_ip2;
\end{verbatim}
description: This function executes the connect statement of Estelle language. The input parameters are the address of the calling module context (_p_ctx1), whose interaction point (referenced by _ip1), is to be connected to the interaction point (referenced by _p_ctx2) of the module instance whose context address is _ip2.

5.3.5 _ipdisconnect:
synopsis: void _ipdisconnect (_ctxa, _ipref)

_DESCRIPTION:

_MODCTXTYYP _ctxa ;
_TYIPDT * _ipref ;

description: This function performs the Estelle disconnect statement "disconnect X.ip" (of one interaction point). The input parameters are the address of the calling module context (_ctxa), and the reference (_ipref) of the interaction point to be disconnected.

5.3.6 _moddisconnect :
synopsis: void _moddisconnect (_ctx)

_DESCRIPTION:

_MODCTXTYYP _ctx ;

description: This function executes the Estelle disconnect statement "disconnect X" (of all external interaction points of a child module). The input parameters are the address of the module instance context (_ctx) whose interaction points must be disconnected.

5.3.7 _detach :
synopsis: void _detach (_ctx, _ip)

_DESCRIPTION:

_MODCTXTYYP _ctx ;
_TYIPDT * _ip ;

description: This function executes the Estelle detach statements either "detach ip" or "detach X.ip". The _detach function is used to detach an interaction point of a module instance. The reference (_ip) of this interaction point is passed as input parameter. The other input parameter is the address of the context of the concerned module instance (_ctx).

5.3.8 _initall :
synopsis: void _initall (_pctx, _p_allctrl)

_DESCRIPTION:

_MODCTXTYYP _pctx ;
_ALLCTRL * _p_allctrl ;

description: This function computes the current list of children module instances of a given module instance. This list has to be explored by further calls to the _getall primitive. The input parameters are the address of the calling module instance context (_pctx) and the address of a control variable used to build the list of children (_p_allctrl).
5.3.9  **_getall_**

**synopsis:**

```c
int _getall (_pctx, _mvarp, _head_key, _p_allctrl)

_MODCTXTYP  _pctx;
_MODVAR     * _mvarp;
int         _head_key;
_ALLCTRL    * _p_allctrl;
```

**description:**

This function successively accesses all module instances of a given header type and belonging to the list previously computed by _initall. The input parameters are:

- a pointer to the calling module instance context (_pctx);
- the address (_mvarp) of the module variable where the result of an access must be placed;
- the numerical unique identifier (_head_key) associated with the header type;
- the address (_p_allctrl) of the control variable used in the call to _initall.

The result of _getall is:

- 0 if no more instances within the computed list,
- not 0 if an instance is selected.

5.3.10 **_rstforone_**

**synopsis:**

```c
void _rstforone (_mvarp)

_MODVAR     * _mvarp;
```

**description:**

This function computes the set of children module instances of a given module instance for a further selection by the _getone primitive. The input parameter is the address (_mvarp) of the module-variable where the result of the call must be placed.

5.3.11 **_getone_**

**synopsis:**

```c
int _getone (_pctx, _mvarp, _head_key)

_MODCTXTYP  _pctx;
_MODVAR     * _mvarp;
int         _head_key;
```

**description:**

This function performs successive access to module instances of a given header type and belonging to the list previously computed by _rstforone.

The input parameters are:

- a pointer to the calling module instance context (_pctx);
- the address (_mvarp) of the module variable where the result of access must be placed;
- the numerical unique identifier (_head_key) associated to the header type.

The result of _getone is:

- 0 if no instance is selected;
- not 0 if an instance is selected.
5.3.12 _setdelay :

synopsis: \[
\begin{align*}
\text{long } & \_\text{setdelay} (\text{ctxp, } t\text{min, } no\text{\_tmax, } t\text{max, } \text{varp}) \\
\_\text{MODCTXTYP } & \text{ctxp} \\
\text{int } & \text{no\_tmax} \\
\text{long } & \text{tmin} \\
\text{long } & \text{tmax} \\
\text{int } & * \text{varp}
\end{align*}
\]

description: This function performs the initialisation of an Estelle timer. The input parameter \text{ctxp} is the address of the calling module context. The input parameter \text{varp} points to the integer which is used to control the timer evolution. This integer is:

- 2 if time is between (now) and (now + tmin);
- 1 if time is between (now + tmin) and (now + tmax);
- 0 if time is after (now + tmax).

If \text{no\_tmax} is not equal to 0 then \(t\text{\_max}\) is meaningless and is assumed to be infinite. The returned value is an unsigned, long integer which is a unique identifier of the initialised Estelle timer. This identifier must never be 0, and serves to cancel the timer, if needed.

5.3.13 _rstdelay :

synopsis: \[
\begin{align*}
\text{void } & \_\text{rstdelay} (\text{delay\_key}) \\
\text{long } & \text{delay\_key}
\end{align*}
\]

description: This function is used to cancel an initiated Estelle timer. The input parameter \text{delay\_key} is the identifier returned by the \_setdelay primitive.

6. AUTOMATIC PROTOTYPE IMPLEMENTATIONS

The C code generated by the Ec compiler is independent of the target implementation platform (hardware + operating system). To obtain an executable code for a given platform, this C code has to be linked to an appropriate implementation runtime library. One such a library is delivered with the EDT package ($\text{SESTEL/estmotc/libe.a}$) for a requested platform. The libraries for the following platforms can be delivered on a special request: Sun-SunOS4.1.3, Sun-SOLARIS5.1.3, HP9000-HPUXB10.20 and PC-LINUX 2.x.

To help generating the executable code the Ec compiler beside generating several C-code files generates also (when \text{-M} option is used) a \text{makefile}. It permits the user automatically compile the C-code generated and link it with the delivered \text{SESTEL/estmotc/libe.a} runtime library and some other user defined libraries (when option \text{-L} was used). This \text{makefile} has to be modified if the runtime library to be linked with the C-code is different from this delivered with the EDT package.

To resume, the user has to invoke the Ec compiler using the following command:

\[\text{ec -M -Luser\_library [other C-code\_generator\_options] source\_file\_name.stl}\]

The above command will generate several C-code files and a \text{makefile} \text{source\_file\_name.mk}.
The Unix command

```
make -f source_file_name.mk
```

issued on a designated platform permits to generate the prototype implementation executable
code (executable file named `source_file_name`).

It is highly recommended to use a devoted directory created within your file system for each
Estelle specification from which a prototype implementation will be generated.

Please note that the C code generated by the Ec compiler for a prototype implementation is a
subset of the C code generated for the simulation. We can thus be confident that the
implementation will reproduce the behaviour verified by simulation

The executable code `source_file_name` generated by Ec compiler (and `make`) when executed
runs as a unique process. The inherent parallelism between modules within specified system is
emulated.

EDT package also provides the means for the distributed implementation in which the executable
codes are generated separately for each sub-system and may be placed on different host
computers within a network. To this end an initial Estelle system specification is `split` into as
many Estelle specifications as there are sub-systems in the original specification. This may be
automatically done by the Splitter (Distributed Specification Generator) tool, a part of the current
EDT package (see the Universal Generator User Manual). Each resulted specification is then
compiled and the C-code generated is compiled and linked with the runtime library of the
designated host platform (as in the case of one-process implementation) and with the additional
communication libraries.

In each of the resulted specifications, a sub-system module from the original Estelle
specification is embedded in a new module. Each external interaction point of the original sub-
system module is attached to an internal interaction point of the new module. The transitions
(including the initialisation "transition") of the new module enable to perform the
communication with the environment, using four generic primitives `mxinit`, `mxwhen`,
`mxoutput`, `mxsend` provided by the EDT universal communication library
`$ESTEL/estmotc/libecm.a`. These four communication primitives are independent of the Unix
communication mechanisms to be used. They in fact call other primitives named `user_init`,
`user_recv` and `user_send` to make a mapping to a given Unix communication mechanism such as
STREAM sockets (TCP/IP) or DGRAM sockets (UDP/IP). The corresponding specialised
communication libraries are `$ESTEL/estmotc/uucplib.a` and `$ESTEL/estmotc/uudpplib.a`. One of
these two libraries has to be linked when `ec` command is invoked.

These primitives should have the following functions:

- `user_init` is called by `mxinit` to:
  - in case of TCP - bind a server sockets to a TCP address and to connect a client socket
to another TCP address
  - in case of UDP or IP - bind a server and client sockets to two UDP addresses
- `user_recv` is called by `mxwhen` when an interaction is to be received by the sub-system
from another sub-system to extract the interaction from the network
- `user_send` is called by `mxsend` when an interaction is to be sent by a sub-system to another
sub-system to put the interaction on the network.

The C-code sources of the above primitives (named `user_init.c`, `user_recv.c` and `user_send.c`) for
TCP/IP and UDP/IP are also delivered within EDT package (in the directories:
`$ESTEL/estmotc/LIBTCP/USERLIB` and `$ESTEL/estmotc/LIBUDP/USERLIB`, respectively).
They may serve as a basis to create similar files for others communication mechanisms (for
example for RAW sockets for IP) the user may wish to use.
Please note that the above 2 stage (universal primitives calling specialised primitives) approach is proposed for the purpose of **easy** prototyping within different environment (TCP/IP, UDP/IP, …) to be done by a non-expert user. For more effective implementation an expert user can modify the sub-system specifications generated by the splitter in order to use directly specialised communication primitives of his choice.

**For the Sun-SOLARIS 5.x platform** the Distributed Specification Generator (splitter) tool also builds executable codes called **supervisor** (an RPC client) and **exec_server** (an RPC server). The **exec_server** has to be executed (once) on each designated host on which at least one of the sub-systems is to be distributed. The **supervisor** starts (sending a corresponding request to the **exec_server**) the execution of the processes corresponding to sub-system modules, on designated computers in the network, according to the initialisation section in the original Estelle specification. The station on which a process has to run must be indicated by an annotation of the INIT Estelle statement that creates the sub-system module instance. Please consult the User Manual of the Universal Generator (Splitter tool) for further details.

The **supervisor** is generated from the INITIALIZE part of the initial Estelle specification composed of several sub-systems. It uses 2 functions (independent of the specification considered):

- **sv_runinit**, which is called for each INIT **annotated** statement within the INITIALIZE part of the initial specification
- **svedt_connect**, which is called for each CONNECT statement within the INITIALIZE part of the initial specification

The C-code sources of **sv_runinit** and **svedt_connect** primitives for TCP/IP and UDP/IP are also delivered within EDT package in the file named **sv_prim.c** (in $ESTEL/estmotc/LIBTCP/SV and $ESTEL/estmotc/LIBUDP/SV, respectively) to serve as a basis to create the similar function for other platforms.

In the above directories there are also the following C-code source files:

- **exec_server.c** (necessary to generate the RPC server **exec_server**),
- **exec_client.c** (the specification independent part necessary to generate the RPC client **supervisor**),
- **exec.x** containing the common knowledge for the RPC client and server (in particular in this file the RPC number to be used is set to the default value 10071972 and the user may change it in case this number is used for other purposes).

These files are given to serve as a basis to create the similar functions for other than Sun-SOLARIS 5.x platforms.

The EDT package contains several facilities to automate the generation of the executable codes for distributed prototype implementations using TCP/IP or UDP/IP Unix communication mechanisms.

These procedures assume that:

- all (distributed) hosts are of the same type i.e., same platform (necessary modifications for the case when hosts are different are very simple) and make part of the same LAN network,
- Ec compiler is devoted to this platform
- a dedicated directory has been created for the implementation purposes
The steps described below have to be followed:

1. Initial Estelle specification modifications
2. Generation of Estelle Specifications for each sub-system and of the files necessary to generate the specification dependent part of the RPC client (supervisor)
3. Generation of the executable code for each sub-system
4. Generation of the RPC client (supervisor) and of the RPC server (exec_server)

1. Initial Estelle specification modifications
   • Each INIT statement within the initial Estelle specification (composed of several sub-systems) concerning a sub-system creation should be annotated by a special comment of the form
     
     \($@\$, "argument_1", argument_2\),

     where argument_1 is the hostname of the host computer on which the sub-system should run and argument_2 is the TCP or UDP port number of this host computer.

     • The current version of the implementation motor does not take into account the timescale specified within the Estelle specification. All delays are interpreted in microsecond. You may have to adjust these values.

2. Generation of Estelle Specifications for each sub-system and of the files necessary to generate the specification dependent part of the RPC client (supervisor)
   
   The splitter tool has to be invoked with the following command (it is highly recommended to create a designated directory for implementation and to issue this command from it)

   \(\text{ug }\text{-S} \text{initial\_annotated\_specification.stl}\)

   As a result several sub-system Estelle specifications \(\text{sub-system}_i\text{.stl}\) and, in addition, 2 files \(\text{SV\_INIT.c}\) and \(\text{SV\_INIT.h}\) necessary to generate the specification dependent part of the RPC client (supervisor) will be generated in the current directory.

3. Generation of the executable code for each sub-system
   
   This step has to be done for each Estelle specification \(\text{sub-system}_i\text{.stl}\) generated during the step 2.
   
   It is highly recommended to generate the executable code for each sub-system in a dedicated directory. Let us call this directory \(\text{sub-system}_i\text{.dir}\).

   We will show below all the necessary elementary steps for just one sub-system \(\text{sub-system}_1\text{.stl}\) and only for the TCP communication mechanism.
- **mkdir** `sub-system_1.dir`
  - **cd** `sub-system_1.dir`

  Creation of a dedicated temporary working directory, named `sub-system_1.dir`

- **ln** `./sub-system_1.stl`

  Copy of the `sub-system_1.stl` Estelle specification generated in step 2 to the `sub-system_1.dir` directory.

- **ec** `-q` `-L$ESTEL/estmotc/libecm.a`
  - `-L $ESTEL/estmotc/utcpplib.a`
  - `-M` `<user-defined options>`
  - `sub-system_1.stl`

  Compilation of the `sub-system_1.stl` Estelle specification and link edit with the universal and TCP communication library. The user may aid some other user-defined options allowed by the `ec` compiler. In particular for System V UNIX system (like Sun SOLARIS 2.x.x) the following options have to be aided: `-L` `-Insl` `-L` `-lsocket`. When the Estelle specification `sub-system_1.stl` contains some `primitive` functions and procedures then the option `-L` `../` has to be aided (`prim.o` stands for the object file stored in the implementation working directory generated from `prim.c` file (`cc` `-c` `prim.c`) and containing the C-coded bodies of the `primitive` functions or procedures).

- **make** `-f` `sub-system_1.mk`

  Compilation of the generated C files using the generated makefile `sub-system_1.mk` to produce `sub-system_1` executable file

- **mv** `sub-system_1..`
  - **cd** `..`

  Move the created `sub-system_1` executable file to the implementation working directory.

The shell-scripts `gentcpsyst` (for TCP) and `genudpsyst` (for UDP) in `$ESTEL/bin` will do all the above steps automatically. They have to be invoked using as arguments:

- user-defined options allowed by the `ec` compiler
- sub-system Estelle specification `sub-system_1.stl`

For example, the following command may be issued (in case you issue this command within the UNIX System V (like Sun SOLARIS 2.x.x) please verify that the shell-script `gentcpsyst` contains the following options `-L` `-Insl` `-L` `-lsocket`, otherwise please aid them in the command line):

```
gentcpsyst -L ../prim.o sub-system_1.stl
```

### 4. Generation of the RPC client (supervisor) and of the RPC server (exec_server)

Both the RPC client (supervisor) and the RPC server (exec_server) generated using the shell script described bellow are adequate ONLY for SunSolaris 5.x platform.

The `exec_server` generated (using the files `exec_server.c`, `exec.x`) is independent of the Estelle specification considered and of the Unix communication mechanism. A major part of the `supervisor` (`exec_client.c`) is also specification independent. The specification dependent part is
generated by the Splitter (files SV_INIT.c and SV_INIT.h). It is highly recommended to
generate both supervisor and exec_server in a dedicated directory. Let us call this directory SV.

The following steps may be done manually to generate both supervisor and exec_server.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• mkdir SV ; cd SV</td>
<td>Creation of a dedicated temporary working directory, named SV</td>
</tr>
<tr>
<td>• cp $ESTEL/estmote/LIBTCP/SV/* .</td>
<td>Copy of files specifying the exec_server (exec_client.c, exec_server.c, exec.x), the specification independent part of the supervisor (sv_prim.c, sv_incl.h) and of the Makefile</td>
</tr>
<tr>
<td>• ln ../SV_INIT.* .</td>
<td>Copy of files specifying the specification dependent part of the supervisor (SV_INIT.c and SV_INIT.h)</td>
</tr>
<tr>
<td>• make</td>
<td>Compilation of the exec_server and of the supervisor using Makefile</td>
</tr>
<tr>
<td>• mv supervisor exec_server .. ; cd ..</td>
<td>Move the exec_server and the supervisor to the implementation working directory</td>
</tr>
</tbody>
</table>

The shell-scripts gentcpsv (for TCP) and genudpsv (for UDP) in $ESTEL/bin will do all the
above steps automatically.

The last steps to run the distributed prototype implementation consist in executing (once) the
RPC server (exec_server) on each host on which one of the sub-system is supposed to run and
to execute the RPC client (supervisor). When the user wish to display on his local host the
messages issued (if any) by the running sub-systems located on the other hosts (such messages
may by the result of ‘printf’ type of qualifying comments inserted in the initial Estelle
specification) then he has to set DISPLAY shell variable on each remote host before running the
corresponding RPC server. The scenario may look like the following:

- rlogin remote_host_name
- cd implementation_working_directory
- ls -l /var/run/supervisor
- xhost + remote_host_name
- ./exec-server
  - exit
  - xhost + remote_host_name
  - ./supervisor

Please note that all sub-systems may run on the same host It will happen when the first
argument of the INIT statement annotations is the same for all such statements concerning sub-
systems within the INITIALIZE part of the initial Estelle specification. Please note that the
second arguments (TCP or UDP port number) of the INIT statement annotations should be all
different. In such a case, it is enough to run the RPC server (exec_server) and RPC client
(supervisor) at the host.

The above method has been used to implement the XTP (version 4.0) protocol. The generated
prototype implementation has been compared with the implementation done directly in C++ by
SANDIA. Both implementations have quit similar properties (almost same performances and
memory occupation). The reader interested in details is invited to read the following publications
[20, 21].
REFERENCES

[20] O. Catrina, E. Lalllet, S. Budkowski, Automated implementation of the Xpress Transport Protocol (XTP) from an Estelle specification, Electronic Journal on Networks and Distributed Processing (EJNDP), Special Issue on CFIP97 (http://rerir.univ-pau.fr/ journal/numero7/)