The Estelle *PEW* User Manual

for Version 2.1

by Graham Wheeler

Data Network Architectures Laboratory

Department of Computer Science

University of Cape Town

Private Bag, RONDEBOSCH

7700

South Africa

Telephone +27.21.650.2662

FAX +27.21.650.3726

E-Mail gram@cs.uct.ac.za or gram@gramix.aztec.co.za

Contents

1 Preface

This manual is a revision of the Preliminary Version of the *PEW* user manual which was released with Version 1.0 of the *PEW*. It has been prepared for the release of Version 2.0 of the *PEW*. In order to highlight the differences between the versions 1.0, 1.1, 1.2 and 2.0, the original introduction has been left unmodified. Instead, supplementary introductions have been added, for each of the successive versions, which summarise changes that have been made. If you are already familiar with the *PEW*, you may read these supplementary introductions immediately; for newcomers, they will be more understandable once you have read the rest of the manual and worked with the *PEW*.

For those who have used the *PEW* before, I hope you like the changes that have been made, and find them useful.

2 Introduction to Version 1.0

The Estelle *PEW* (*P*rotocol *E*ngineering *W*orkstation) is a tool for the development, testing and simulation of protocol specifications written using the Estelle (Extended State Transition Language) Formal Description Technique (FDT). The PEW provides an integrated environment complete with screen editor, compiler, linker, and debugger/interpreter all in a single package.

The Estelle language was developed by the International Organisation for Standardisation (ISO) as one of several techniques for protocol specification. The *PEW* conforms closely to the Draft International Standard DIS 9074, promulgated in July 1987. The major restriction of the *PEW*'s implementation is the lack of a real (that is, floating point) data type.

This document is the first version of the Estelle *PEW* User Manual. It describes the first general release of the *PEW*, which is Version 1.0. All previous releases were internal to the Department of Computer Science of the University of Cape Town.

The *PEW* really began in 1987, when Jaque van Dijk, under the supervision of Professor Pieter Kritzinger, developed a compiler for Estelle. This compiler was completed in late 1988, at which point Graham Wheeler took over the project. Inspired by the 'Integrated Environment' language development systems being marketed by the Borland company (in particular, Borland's TurboC, which was used as the development language for the *PEW*), Graham decided that rather than just developing a stand-alone interpreter for Estelle, an integrated development environment would be built.

The first component to be developed was the editor, which was written during the early part of 1989. Once this was completed, Graham attempted to integrate Jaque's compiler into the editor. The obstacles to this proved insurmountable: the compiler had been written in Pascal, and was also sufficiently unstable for it to be executed indirectly from the editor. In April 1989, Graham wrote a Pascal to C translator, and by May had the 6 500 lines of Pascal translated to just under 6 000 lines of C. Over the course of 1989 the compiler was refined, large parts of it were rewritten, and many bugs were removed. The final product is a stable compiler which is 20% smaller (the executable code is 40% smaller), and which compiles Estelle code at about 5 000 lines per minute (about three times faster than the original) - a true demonstration of the power of C!

Hand in hand with the redevelopment of the compiler went the development of the interpreter. The compiler generates a (mostly) low-level code called E-Code, based on Pascal's P-Code. The demands of interpreting Estelle led to several rethinks of aspects of the E-Code, which in turn led back to changes in the compiler. The two components were thus inseperable in design, although they were kept as seperate programs until July 1989 when the first integrated version was produced. At this stage the debugger was primarily a low-level debugger, allowing examination of the E-Code, stepping through E-Code, examination of process's internal stacks, and other details which have subsequently been hidden (but not removed).

Version 0.6 of the *PEW* was released internally within the Department of Computer Science at U.C.T. in August, which provided valuable feedback about the

system. Many thanks to the senior students in the Department who unwittingly did the Beta testing!

The current version of the *PEW* incorporates a new user interface called the *Process Browser*. This is a multi-windowed environment which allows the user to constantly view details of the current executing process, or to examine other processes freely. This environment makes it easy to set breakpoints, view connection endpoints of channels, delete interactions from queues, examine the Estelle source code, view statistics about the firing of transitions and use of channels, and so on. Context-sensitive help is available at all times. We believe this user-interface to be one of the most exciting features of the *PEW*.

The fact that the *PEW* gathers execution statistics as it operates makes it very easy to use as an execution profiling tool for protocol specifications. For example, a user may calculate the throughput and error rate of a protocol under various loads very quickly and easily, without having to add explicit code to the Estelle specification to gather statistics. The *PEW* also detects and reports deadlock when it occurs, and can tell when there are interactions queued which can never be dealt with. It thus aids the correctness testing of protocol specifications.

This version of the *PEW* was demonstrated at Forte '89. The *PEW* continues to be developed. The next version is scheduled for the end of 1990; we hope that we will be able to largely eliminate the need for the user to perform multiple executions in order to evaluate performance under different conditions. Our aim is to incorporate an analytical tool into the *PEW* which can generate and solve queueing models of protocol specifications.

We hope that you enjoy using the *PEW*. If you have any comments or suggestions to make, or bugs to report, please make use of the evaluation form included in the Appendices and return it to us.

3 Introduction to Version 1.1

Version 1.1 of the *PEW* was released in February 1990. It contains a number of changes, which are summarised below:

- A number of bugs have been fixed (of course).
- The *F10* key is now used to quit the interpreter/debugger, and the two different execution modes (*Animate* and *Execute*) have been combined into a single mode serving both purposes, assigned to the *F9* key.
- Three new functions have been added: `QLENGTH`, which returns the length of a reception queue when applied to an interaction point; `FIRECOUNT`, which returns the number of times the containing transition has been fired, and `GLOBALTIME`, which returns the current (simulation) time.
- The arrow on the left of the screen indicating which source line is being executed changes to an 's' when the scheduler is running.

- Output procedures WRITE and WRITELN also write to the log file if logging is on.
- The destination of output from WRITE and WRITELN can be controlled using a compiler directive.
- The pseudo-random number generator can now be seeded by the user, to replicate executions.
- Semaphores are used to prevent the interpreter being executed during critical sections; previously infinite recursions were possible if the interpreter tried to dump statistics to the log file when the process tree data structures were in a fluid state.
- The process browser windows can now be changed in size (to a limited extent) using *Shift-F1* and *Shift-F2*.
- A low-level E-code display can be invoked from the process browser with Alt-F10; this is of relevance only to those who are interested in the internal implementation of the *PEW*.

4 Introduction to Version 1.2

In March 1990, Version 1.2 of the *PEW* was completed. The enhancements made in this version include:

- Some reachability and deadlock analysis of state transitions and interprocess communication has been added.
- As a result of the previous change, the compiler can also now issue non-fatal warnings.
- Pressing *F1* or *Alt-H* when an error or warning window is displayed causes help for that error message to be displayed. The help is contained in a plain text file `ERROR.HLP`, which may be modified.
- An *ANALYZE* menu has been added to the Editor/Compiler. This menu allows an independent variable to be specified, together with a range through which it should be varied. The *PEW* can then automatically recompile and reexecute the specification for each value of the independent variable (traeting it as a top-level `CONSTANT` definition). The user may specify expressions to be evaluated and tabulated as a result of this process. This makes it easy to produce tables of, for example, throughput and error rate of a protocol against timeout.
- `ANY` constants are now supported.
- The `TERMINATE` statement, a new addition to Estelle, has been implemented.

- The syntax of transitions now agrees fully with the standard, and nested transitions are now supported.
- All forms of the `DETACH` statement are now implemented.
- `DELAY` clauses can now have either uniform, Poisson or exponential distributions, selected by a compiler directive.
- An environment variable, `PEWPATH`, can be set to allow the *PEW* to find its help files even if run from a different directory.

Thus, apart from the lack of a `REAL` type, the only areas where the *PEW* is still deficient from the standard is that ... types are still not allowed (and never will be, while the *PEW* remains execution-oriented); `ANY` constants are handled by 'cheating' somewhat), and the `ANY` clause is not supported, but will be soon (as soon as I can figure out an elegant way of doing it, instead of the hack methods that I've thought of so far...).

5 Introduction to Version 2.1

Version 2.0 of the *PEW* was completed in August 1991, with additional changes since then bringing the latest version (May 1992) to 2.1. The number of additions is considerable, and includes:

- Further compiler directives giving increased control over inter-process communication, delays, and I/O;
- The system has been split into stand-alone versions of the compiler and interpreter, as well as the integrated version. The stand-alone components can handle larger specifications;
- The stand-alone interpreter can perform a 'transition sequence analysis' of some systems, and produce a graph specifying all possible sequences of transitions under different probabilities of message loss and delay values;
- This graph can be checked for reachability by a separate reachability checker, which can detect problems such as livelock and deadlock;
- The graph can also be processed by an additional utility which can predict transition throughputs for different parameter values (that is, `OUTPUT` loss and `DELAY` values).
- The statistics output by the *PEW* are, in some cases, more accurate, and are more comprehensive. We have used these statistics to build performance models based on imbedded Markov chains. A utility for solving such models is also included.

- A trace of interaction outputs between the various processes can be produced by turning on an option in the Options menu of the *PEW*. This trace is written to a text file named `TRACE`.

Unfortunately, the `ANY` clause is still not supported. The *PEW* has been used primarily with specifications of point-to-point protocols, so this clause has not been needed.

Many of the additions were made as part of my Ph.D. thesis which examined the problem of performance analysis and prediction from formal specifications (specifically Estelle specifications). A modelling method based on imbedded Markov chains was employed. Tools to solve these models, and generate them automatically in some cases, are also included in this distribution. These tools are included in three separate `.ZIP` files, together with basic documentation.

6 Release Contents and Installation

The *PEW* Version 2.1 distribution includes contains the files:

- `pew.exe` – the editor, compiler and interpreter
- `pew_help.txt` – the *PEW* Help Text file
- `pew_help.idx` – the *PEW* Help Index file
- `error.tbl` – the *PEW* error and warning message table
- `error.hlp` – the *PEW* error and warning help file
- `template` – the Estelle language template definition file
- `langhelp` – the Estelle language help file
- `pewdemo.dat` – a script file containing commands treated as keypresses by the *PEW*, for a quick demonstration
- `demo.bat` – a DOS batch file to execute the *PEW* demo
- `abs.est` – a specification of the Alternating Bit protocol, used by the demo
- `ec.exe` – the stand-alone compiler
- `ei.exe` – the stand-alone interpreter
- `reach.zip` – the reachablity checker, with example
- `predict.zip` – the performance predicter, with example
- `solve.zip` – the model solver, with example
- `epretty.exe` – an e-code pretty-printer
- `userman.tex` - the LaTeXmanual source

Additionally, the disk contains a zip file called `test.zip`, which contains a number of small Estelle specifications that have been used to test different aspects of the *PEW*.

These files should all be copied into the directory in which you intend developing your Estelle specifications. If you run the *PEW* from a directory other than that in which it is installed, you should set the environment variable `PEWPATH` to contain the drive and directory path where the *PEW* and its related files can be found. Consult your DOS manual for details on setting environment variables if you are unfamiliar with this technique.

A program called `unzip.exe` and a batch file called `install.bat` is bundled with the *PEW* distribution on floppy disk. For `ftp` distribution, the files are all in one zipfile, so it is assumed you already have `unzip` or its equivalent.

7 Using the *PEW*

In order to use the *PEW*, a specification must first be entered, using the *PEW* editor or some other editor. The *PEW* compiler must then be used to compile the specification. If an error occurs during compilation, compilation stops and control is returned to the editor. The cursor is positioned at the offending location and the error is reported on the message line at the bottom of the screen. When the error occurs, it is also reported in an error window, which can be cleared by pressing any key. An exception to this is pressing *F1* or *Alt-H*; these keys will cause the *PEW* to look up the help text associated with the error or warning, and display it.

Once a specification has been successfully compiled, it may be executed using the interpreter. Each of these components will be discussed in turn in the following sections of the manual.

The file DEMO.BAT will cause the *PEW* to read 'keystrokes' from the file PEWDEMO.DAT, and give a short demonstration of some of its features.

8 The *PEW* Editor

Upon invoking the *PEW*, which you do by simply typing `PEW`, you are shown a welcome message window. Pressing any key will remove this window and place you in the *PEW* editor. The screen consists of a main menu bar running along the top, an edit window occupying the bulk of the screen (including an information line at the top), and a message line at the bottom of the screen.

If you started the *PEW* without a file argument, it will attempt to open a file called `NONAME.EST` and load it. Otherwise it will attempt to load the file that you specified. For example, to start the *PEW* and load the file `ABS.EST`, you can use the command `PEW ABS.EST`. If the *PEW* cannot load the file you specified (or `NONAME.EST` if you didn't specify any file), the editor buffer displayed in the editor window will be empty, but will be associated with the corresponding name.

It is possible to save your editor configuration explicitly at any time, or automatically when you exit the *PEW*. If you do this, and you invoke the *PEW* without a file argument, it will attempt to load the file that was loaded when the configuration was last saved. The editor configuration consists of the key assignments, keystroke macros, option menu settings, and filenames.

Once in the editor, you may simply type text. Pressing *F1* or *Alt-h* at any stage will give you context-sensitive help. Holding down the Control (*Ctrl*) key and pressing the capitalised letter of a main menu option will result in that option being invoked from the main menu. This may result in a submenu popping up. For example, *Ctrl-F* will invoke the *File Menu*.

When you are in a menu, you may move around with the cursor keys or by pressing the letter corresponding to a menu option (for example, *L* for the *Load File* command in the File Submenu). Pressing *ENTER* will select that menu option. In some cases this will result in a prompt on the message line (for example, *Load File* will prompt you for a file name). Sometimes a default value will appear after the prompt. You may edit this default or your own input using the cursor keys, Del, Backspace, Ins (to toggle insert/overstrike mode) and ESC (clear value). When you are satisfied press *ENTER*. To leave a menu without selecting an option use *ESC*.

We will now consider the various menus and commands available from within the Editor.

8.1 The Main Menu

The main menu is displayed on the top line of the screen. It is accessed from the editor by holding down the *Ctrl* key and pressing the uppercase letter corresponding to the menu item required. If one of the two menus (*File* or *Options*) is invoked, and subsequently exited by pressing *ESC*, you will remain on the main menu but the menu will close. You can then move around the menu with the left and right cursor keys, and press *ENTER* to select a menu item. To get back to editing, select the *Edit* item. The main menu items are:

Ctrl-F Invoke the File Submenu (see Section 8.2).

Ctrl-E Resume editing.

Ctrl-C Compile the Editor File.

Ctrl-X eXecute the Editor File, compiling it first if necessary.

Ctrl-O Invoke the Options Submenu (see Section 8.4).

Ctrl-A Invoke the Analyze Submenu (see Section 8.10).

8.2 The File Submenu

The File submenu contains commands primarily pertaining to the manipulation of files, such as loading and saving files to and from the editor. The items available are:

Load File Specify a file to load from disk into the editor buffer.

Load Log Load the execution summary file 'ESTELLE.LOG' from disk. See Section 11 for details.

Save Save the edit buffer to disk using the current *Save Name*.

Change Name Change the current *Save Name*.

New Clear the Editor buffer.

Goto Line Move the cursor to a specified line in the edit buffer.

OS Shell Run a DOS shell. Type "exit" to return from DOS to the *PEW*.

Quit Quit to DOS. You may also use *Alt-Z* to quit.

Read Scrap Read a disk file into the scrap buffer. The scrap buffer is used for 'cut and paste' editor operations.

Write Scrap Write the scrap buffer contents to a disk file.

8.3 The Compile and Execute Commands

These two main menu options allow you to compile you Estelle program, and invoke the interpreter. The *Compile* command will simply attempt to compile your specification. If a compilation error occurs, this will be reported in a pop-up error window. You may press any key to remove this window (other than *F1* or *Alt-H*, which will cause help pertaining to the error to be displayed); the error message will then be shown on the message line and the cursor positioned where the compiler encountered the error.

Error messages contain a text message, a line number, and an *error code*. Error codes consist of an 'E' followed by a three-digit number. These error codes are used

internally to identify the place in the *PEW* where the error occurred, and can be ignored by the user.

The *Execute* command will only compile the program if it has not already been compiled. If compilation is unsuccessful, then you may proceed as if you had just selected *Compile*. If compilation is successful, after pressing a key to clear the Compile window (if applicable) you will be placed in the *PEW execution environment*. This will be dealt with in detail later.

8.4 The Options Submenu

This menu contains various options which may be set according to your preferences for the editor; for example, how lines that are too long to be displayed on a single screen line are to be treated. It also is where you enter search and replace text strings for these editor commands. The available items are:

Tabsize — Specify the default tab stop setting. The *PEW* editor handles tabs in a rather primitive fashion. Whenever the TAB key is pressed, or a file is loaded, all tabs are expanded to spaces using the current tabsize setting. When you save a file, this process may be reversed depending on the *Fill Character* option (see below). Thus while using the editor, all whitespace is effectively made up of space characters, not tabs. To make this slightly less unwieldy, the backspace key can be used with a Shift key to delete all whitespace up to the previous tab stop.

Backup Files — If this option is on, the *PEW* will rename the old copy of a file before saving a new copy. The old backup copy will have the same name but a '.BAK' extension.

Fill Character — If this is set to Tabs, then spaces will be compressed into tabs wherever possible when a file is saved to disk, using the current *Tabsize* setting. See *Tabsize* for more details.

Wrap Mode — Switch between line wrap mode or horizontal scroll mode. This affects the treament of long lines. When wrap mode is on, long lines are 'wrapped' over when necessary onto the next screen line. When wrap mode is off, long lines are truncated at the left and right edit window edges, and the entire display is scrolled left and right as necessary.

Left Search — This lets you specify a set of characters to be used by the character set matcher. Whenever *Ctrl-PgUp* is pressed, the cursor will be moved backwards to the first character in the edit buffer that matches any character from the specified set. This can be used as a simple parenthesis matching facility.

Right Search — This is equivalent to the *Left Search* option described above, except that the cursor is moved forward in the file rather than backward.

Search Text — Allows you to specify the current search text for the search and translate commands.

Translate Text — Allows you to specify the current replacement text used by the translate command.

Case Sensitivity — Lets you toggle case sensitivity on and off for use in searching and translating.

Identifiers Only — Lets you toggle identifier matching on and off for use in searching and translating. When this option is on, the search text must correspond to text in the edit buffer which is surrounded by whitespace or punctuation only. This is useful is you wish to, for example, change all occurences of the identifier 'alternating_bit' to 'AB', but you don't want to change any occurences of the identifier 'alternating_bit_body' in the process.

Autosave Config — When this option is on, the editor configuration is saved automatically upon exit. The configuration is saved in the file *PEW.CFG*.

Save Config — Save the configuration immediately. The editor configuration consists of the current file name and position, option menu settings, and key assignments and macros.

Random Seed — This option allows you to specify a seed for the pseudo-random number generator (the seed is usually chosen randomly). It allows you to replicate an execution, even though executions are, in principle, nondeterministic.

Generate Trace — If this option is on, the *PEW* will produce a file called `trace` when the specification is executed. This file shows the sequence of interactions that are exchanged between processes over time.

8.5 Fixed (Non-Reassignable) Commands

At present, all editor commands are non-reassignable; however, all that is lacking is the ability to edit the default key assignments. The commands listed in this section are those whose key assignments are "hard-coded" into the editor, and cannot be changed even when a key-assignment editor has been written.

↑ — Move the cursor up one line.

↓	Move the cursor down one line.
←	Move the cursor left one character.
→	Move the cursor right one character.
Home	Move the cursor to the start of the current line.
End	Move the cursor to the end of the current line.
PgUp	Scroll the screen up.
PgDn	Scroll the screen down.
Del	Delete the character under the cursor.
Backspace	Delete the character immediately before the cursor.
Shift-Backspace	Delete all whitespace characters up to the previous tab stop setting.
F1 or *Alt-H*	Provide help.

8.6 Other Cursor Movement Commands

This section lists the remaining cursor movement commands that are available. Remember, you can use the *Goto Line* command on the File menu to position the cursor on a specific line.

Ctrl-←	Move cursor left one word.
Ctrl-→	Move cursor right one word.
Ctrl-Home	Move cursor to the start of the file.
Ctrl-End	Move cursor to the end of the file.
Ctrl-P	Set a bookmark. Up to ten positions in the file can be remembered with this facility, and recovered with the *Goto last bookmark* command described below.
Ctrl-G	Goto last bookmark. This command restores a remembered cursor position that was saved with the *Set a bookmark* command described above. Positions are restored in the opposite order to which they are saved, and the bookmark list is cyclic.

Figure 1: An example of a Normal block

8.7 Block Commands

This section describes commands used to cut and paste blocks of text.

Blocks can be cut and pasted in two forms: *normal* blocks and *column* blocks. Normal blocks are simply contiguous character sequences from the file, for example, all text from line 10 column 13 to line 17 column 2 inclusive. Column blocks are columnar substrings of line ranges, for example, columns 10 through 20 of lines 10 through 19 inclusive. There are commands for marking blocks of either type, and for pasting blocks of either type. The two types can be combined, that is, a block can be cut as a normal block and pasted as a column block, and vice-versa.

Blocks are delimited by firstly dropping a *mark* of the appropriate type at the one extreme, and then moving to the other extreme position. The block will be highlighted during this process. The block may then be cut or copied to the scrap buffer, from where it may be pasted.

It is also possible to save the contents of the scrap buffer to a file, or load the contents of a file into the scrap buffer. These operations were described in Section 8.2.

Alt-M	Set a normal mark.
Alt-C	Set a column mark.
Alt– (Alt-Minus)	Copy marked text to scrap buffer.
Del	Delete marked text, copying to scrap buffer.
Ins	Paste scrap buffer contents at cursor position as a normal block.
Alt-P	Paste scrap buffer contents at cursor position as a column block.

8.8 Search/Translate Commands

The search/translate commands provide the means to loacte specific text or characters, and to replace occurences of some text string with a different text string. See Section 8.4 for details on how to specify the search and translate text patterns, and character match sets.

Alt-S	Search forward from cursor position for search text.
Ctrl-S	Search backward from cursor position for search text.
Alt-T	Translate text which matches the search text to the specified translate text.
Ctrl-PgUp	Perform left character match.
Ctrl-PgDn	Perform right character match.

8.9 Miscellaneous Commands

Alt-Z Quit the *PEW* and return to DOS.

Alt-U Undo all changes that have been made to the current line since moving the cursor onto this line.

Alt-R Record a keystroke macro and assign to key. Keystroke macros can be assigned to the keys Alt-0 through Alt-9. At present, they may only be used in the editor, not in the execution environment. After pressing Alt-R, you will be asked to press the key to which you are assigning the macro. All keypresses will be recorded until the macro key is pressed again. Subsequent presses of the macro key will insert the macro characters into the keyboard buffer, where they will behave as though they had been pressed.

Alt-D Delete the current line.

Alt-F1 Call up language help for the Estelle keyword at the current cursor position, if any.

Ctrl-F1 Perform template expansion on the Estelle keyword at the current cursor position, if any.

The last two commands make use of the 'template' and 'langhelp' files. Both of these files may be freely edited, allowing you to create your own help and template details. In each case, the word at which the cursor is positioned is used as a lookup index into the appropriate file; if an entry is found, it is displayed (in the case of help) or it replaces the index word (in the case of template expansion). The file formats consist of alphabetically sorted entries, where each entry begins with a index line consisting of an '@' character followed by the string which is used as the match index, and then a number of lines containing the help or replacement text. For the help file, the tilde character '~' can be used to indicate the start of a new line. You must maintain the alphabetical ordering of the indices, as a binary search method is used on these files and they are assumed to be sorted.

8.10 The Analyze Submenu

The Analyze menu provides you with a way to execute a specification a number of times, varying a single parameter (called the *independent variable*) through a range of values. Up to eight *dependent expressions* can be entered using an expression editor, and the values of these expressions will be tabulated in the *PEW* log file.

The use of the Analyze menu is best illustrated by example. We shall use the sample specification of the Alternating Bit protocol, to calculate the throughput and error rate of this protocol as a function of the timeout. We will vary the timeout from 3 seconds to 6 seconds, in steps of one second each. For each timeout, we will excercise the protocol for 15 minutes of simulated time (9000 seconds).

The throughput and error rate of the protocol depend on the total time, and the execution counts of two transitions: the *timeout and retransmit* transition, and the *received acknowledgement* transition. The first represents the number of unsuccessful transmissions, while the second is the number of successful transmissions. In the specification, these transitions are named *Retrans* and *GotACK* respectively. This gives us the equations:

- $Thruput = GotACK/time$
- $Err_Rate = Retrans/(Retrans + GotACK)$

We now have sufficient background to perform the analysis. The steps involved are as follows:

- Load the `abs.est` specification into the editor.
- Comment out or delete the line in the constant definition section which specifies `Timeout=4`. This is necessary as the analysis process will be defining this constant for us.
- Press `Ctrl-A` to invoke the analyse menu.
- Move the highlight to *Independent Variable* and press *ENTER*. You will be prompted to enter a variable name. Enter `Timeout`.
- Move the highlight to, select and enter the values 3, 6 and 1 for the menu options *From Value*, *To Value* and *Stepsize* respectively.
- Select the *Edit Dependents* option. A new window will appear on the screen; this is the *expression editor*. It works similarly to the usual editor, except that the *ESC* key clears the current line and the *RETURN* or *ENTER* key returns to the menu. The cursor keys, *Ins*, *Del* and *Backspace* all work as usual. Enter the expressions for throughput and error rate as they appear above. The general rules for expressions are that (i) an expression must have a unique identifier on the left hand of the assignment, and must only refer to transition names and the predefined identifier *time* on the right hand side, and (ii) an expression may not be split over more than one line. Expressions are only parsed after all the executions are complete, so it is often wise to do a quick test run with a single value of the independent variable. Badly parsed expressions, or those that involve division by zero, have their results indicated as a string of hashes (####). In Version 1.2 of the *PEW* expressions may not refer to other expressions, and are not saved when the *PEW* is quit.
- Select the *Analyze* option. You will be prompted for an execution timelimit. Enter 9000.

The *PEW* will now compile and execute the AB specification for each value of `Timeout`, tabulate the results of all named transitions, and the results of all dependent expressions. We can then examine the log file, and see how our timeout setting affected the throughput (measured in frames per second) and error rate (a proportion).

In the case of the AB specification, there are two instances of AB protocol processes executing. The *PEW* keeps track of the number of instances of a process as well, and uses the mean value of the execution count when evaluating expressions. In other words, if the first AB instance had 47 retransmissions, and the second had 51, the *PEW* will report and use the value 49.

9 The Execution Environment

We now turn to the interpreter and its user-interface, which we refer to as the *PEW Execution Environment*. This is a separate environment to the editor environment in that files cannot be edited in the execution environment, and the main menu bar is inaccessible, with the exception of the *Edit* command which returns to the editor environment. The latter can be done by pressing *Ctrl-E* as before, or alternatively *Alt-Z*.

9.1 The Implementation of Time

Before we consider the user-interface to the execution environment, it is appropriate to explain how process scheduling and the handling of time has been implemented in the *PEW*. We distinguish between the *simulation time* and the *scheduler iteration*. Execution proceeds systematically iteration by iteration until all possibilities are exhausted - only then are `DELAY` clauses examined to determine if progress is possible in the future or whether deadlock has occurred. If all that is preventing the *PEW* from proceeding is a `DELAY` clause, the *simulation time* is increased by the smallest delay found. This process continues indefinitely until deadlock occurs or the user terminates it.

`DELAY` clauses are handled as follows: in the case where only one argument is specified, that argument is used as the delay value. Where two arguments are specified, a delay value between the two is chosen (arbitrarily, according to the specified distribution) as the actual delay. Within a specification, compiler directives can be sprinkled specifying the delay distribution to be used. The default is a uniform distribution, selected with the directive {$U}. Also available are Poisson distributions, geometric distributions, and an exponential distributions. Each of these requires an argument specifying the expected value, and they have the forms {$P*val*}, {$G*val*} and {$E*val*} respectively.

As time only progresses when `DELAY` clauses are present, the *PEW* differs from the Estelle standard, which specifies that time must eventually advance. A specification with no `DELAY` clauses will never advance beyond time 1 in the *PEW*, as all transitions without `DELAY`s are regarded as requiring no time to execute. It was felt that this was a more appropriate implementation as it affords the user more control over the handling of time.

At each iteration, every transition of every active process is scrutinised to see if it is enabled. Subsequently we iterate through each active process and execute any selected transitions. The process thus consists of:

- Step 1: Evaluate all transition clauses
- Steps 2-n: Iterate recursively through each active process, executing a transition if appropriate

9.2 Controlling Execution

The first step described above is regarded as atomic and non-interruptable in the *PEW*. It is only possible to step execution through transition bodies, *not* their clauses. However, the *PEW* provides tabular information about clause evaluation, so that the results of this process can be examined. For controlling the execution of a specification, several alternatives are available.

It is possible to single-step execution in the *PEW* on three different levels: the level of Estelle statements, the level of transition blocks, and the level of scheduler iterations. This is achieved using the *F2*, *F3* and *F4* keys respectively.

Execution may also be *animated*, that is, execution may proceed automatically with a user-specified delay between the execution of each Estelle statement. To select this option, use *F9*. You will be prompted to enter an inter-statement animate delay in units of milliseconds. Entering a delay value of zero (the default) specifies *Fast Execution*. This is useful when you wish to quickly execute to a certain time or breakpoint, or are only interested in the execution summary. Unlike animation where full screen updating is constantly performed, during fast execution only the time display is updated. This leads to a great speed improvement in execution. After specifying the delay, you will be asked to specify a time at which execution is halted and control returned to the user. If you enter a value of zero (the default), execution will continue until an interruption or breakpoint.

Note that the timing routine used for animation delays is ill-behaved on some hardware; if you select animation and nothing seems to be happening, wait for about 15 seconds and the animation should begin.

Both animation and fast execution may be interrupted at any time by a keypress. When memory runs low, the *PEW* will not allow fast execution to occur, but will always switch back into single-step mode.

9.3 The Process Browser

Within the execution environment, the screen display changes considerably from the editor environment.

The first thing to notice is that the edit window is still present, but now only displays five lines of the Estelle source. Three new windows have been added to the screen; these, together with the message line at the bottom of the screen, constitute the *Process Browser*. The windows are entitled *Children*, *Transitions*, and *Interaction Points*. The sizes of the new windows may be altered by using *Shift-F1* and *Shift-F2*. The size of the edit window is fixed.

At any stage, there is a single *current active process*. During execution, this process is the same as the *currently executing process*; however, during browsing, the user may make any process the current active process without affecting the currently executing process. The Process Browser always displays information pertaining to the current active process, specifically its module variables (in the *Children* window), its transitions (in the *Transitions* window) and its interaction points (in the *Interaction Points* window). The attribute, module variable name, module body

type and current state of the current active process are displayed on the message line. A two character abbreviation is used for attributes, namely:

- Un - Unattributed
- SP - Systemprocess
- SA - Systemactivity
- Pr - Process
- Ac - Activity

In the case of the specification itself, the name of the specification is displayed as the module body type, while the keyword 'Specification' is displayed in place of the module variable name.

At any stage there is a *current active window*. This window contains a highlight bar, which can be moved up and down with the cursor keys. The highlight bar is used to select items (be they transitions, module variables, source lines or interaction points) upon which actions may be taken. Only non-empty windows may be active. To switch from one window to another, the *F6* key is used. Within each window, pressing *ENTER* results in some action, usually the popping up of a menu. Each window will be considered in turn.

9.3.1 The *Source* Window

The source window is the original edit window. The name *Source* window rather than *Edit* window is to emphasise the fact that it is not possible to edit the file during execution. This window allows the user to examine the Estelle source code, and to track execution when single-stepping on Estelle statements.

The source window indicates the currently executing source line with an arrow on the left-hand border of the window. This arrow may or may not be visible, depending on whether the executing line is visible within the window or not (if not, it is because the highlight bar has been moved away from it; the highlight bar is always visible in the active window and the window contents are scrolled if necessary to allow this). When the scheduler is running, this arrow changes to an 's'.

Pressing *ENTER* inside the source window invokes the *Line Menu*. This menu has three commands: *Set Breakpoint*, *View Breakpoints* and *Goto Line*.

The *Set Breakpoint* and *View Breakpoints* commands are common to the other window menus as well, although the type of breakpoint created by the *Set Breakpoint* command varies depending on the window. For the source window, breakpoints are set on Estelle source lines. Before each line is executed, a check is made to see if a breakpoint has been set on that line, and if so, this breakpoint is processed before the line is executed. The details of breakpoint setting and the *View Breakpoints* command will be dealt with in Section 10.

The remaining command, *Goto Line*, is identical to the editor version; you will be prompted to enter a line number, and the highlight will subsequently be positioned at that line.

Remember, help is available at any stage by pressing *F1*.

9.3.2 The *Children* Window

This window displays the module variables of the current active process. It should be noted that the *PEW* gathers much of its symbolic information during execution of a specification. This has several implications, amongst them the fact that it only knows that a process has module variables once it begins to execute that process, and that it only knows the names of module variables once they have been initialised. Uninitialised module variables are simply dsplayed as having the name 'Uninitialised'.

Pressing *ENTER* on a highlighted module variable will make the process associated with that module variable the current active process. In this way it is easy to move down the process hierarchy. To move up this hierarchy, the *F5 Up Level* key is used; this activates the parent of the currently active process. When a different process is activated, all the windows of the process browser will change to reflect the new active process.

By using the *F5* key in conjunction with the *Children* window, it is a simple matter to examine any process.

9.3.3 The *Transitions* Window

This window shows the transitions associated with the current active process. In each case, the priority is displayed (if there is one), followed by the *from* and *to* states of the transition. These are followed by details about the clause evaluation of each transition, namely the state of the *Provided*, *When* and *Delay* clauses, whether the transition was *Enabled*, and whether it has been *Selected* for execution. Each of these is identified by an appropriate letter, namely *W, P, D, E* and *S* respectively. In each case the convention used is 0 for false, 1 for true, and a – for not applicable (in other words, no such clause exists for that transition).

Furthermore, the transition window will indicate the currently executing transition with an asterisk on its left. If the current active process is not the current executing process, no asterisk will be seen.

Pressing *ENTER* on a highlighted transition will invoke the *Trans Menu*. This menu includes the *Set Breakpoint* and *View Breakpoints* commands mentioned earlier, which are discussed in Section 10.

The remaining command in the *Trans Menu* is *View Stats*. This command displays a table of statistics associated with the currently highlighted transition. This table contains the fields:

From The *from* state of the transition;

To The *to* state of the transition;

Enabled A count of the number of times the transition has been enabled for execution;

Fired A count of the number of times the transition has been fired or executed. This may differ from the previous figure, as being enabled does not necessarily imply being selected for execution;

First The time at which the transition was first enabled;

Last The most recent time when the transition was enabled;

Back The mean time elapsing between this transition being executed and the transition just preceding it being executed;

Self The mean time elapsing between successive occurences of this transition;

Forward The mean time elapsing between this transition being executed and the transition just following it being executed.

In the case where multiple *from* states where specified in the transition, the *from* state will be displayed as a number of periods '...'.

9.3.4 The *Interaction Points* Window

This window shows the names, reception queue lengths and reception queue contents of the interaction points of the current active process. It is possible to scroll through a long queue by highlighting its interaction point and using the left and right cursor keys to move forwards and backwards respectively. The displayed length will change to reflect the length of the queue from the first *displayed* interaction to the end of the queue.

By pressing the *DEL* key, the first *displayed* interaction in the currently highlighted queue can be deleted. Confirmation of the deletion will be requested before it is performed. This provides a way to test how a specification handles lost interactions without having to write Estelle code to randomly lose interactions.

Pressing *ENTER* invokes the *IP Menu* for the currently highlighted interaction point. As for the *Line Menu* and *Trans Menu*, the *Set Breakpoint* and *View Breakpoints* commands are once again available. Two other commands are available: *View Stats* displays a statistics table for the currently highlighted inetraction point, while *View CEP* will find the connection endpoint of this interaction point, and activate its containing process, as well as highlighting the appropriate interaction point. This latter command provides an easy way to examine the statistics for interactions output through an interaction point: the statistics and queue displayed are for reception only, thus by viewing the reception statistics of the connection endpoint of an interaction point, the transmission statistics and queue can be seen. These are complicated in the case of common queues or complex connection and disconnection sequences but these are rare in practise.

The statistics table for an interaction points contains the fields:

Name The name of the interaction point;

Total Traffic The total number of interactions dequeued through this IP due to the execution of `WHEN` clauses;

Mean Length	The average length of the reception queue of this IP;
Max Length	The longest length ever reached by the reception queue of this IP;
Mean Time	The average length of time spent by an interaction in the reception queue before being dequeued;
Max Time	The maximum length of time spent by an interaction in the reception queue before being dequeued.

10 Breakpoints

The *PEW* allows breakpoints to be set on line numbers being reached, transitions being enabled or executed, and interactions being output through interaction points. The type of breakpoint depends on which window is active when the breakpoint is set; breakpoints are always set on the currently highlighted item (line, transition or interaction point) in the current active window. For transition breakpoints, the user must also specify whether the breakpoint is on *enablement* or *execution* of the transition.

Breakpoints may include *pass counts*, specifying a number of times that the breakpoint conditions must hold before the breakpoint *matures* or fires. The default pass count is zero.

Breakpoints may be *active* or *suspended*; suspended breakpoints do nothing and are not checked, but may be reactivated later explicitly by the user, or by the action of some other breakpoint. The other action that may be taken by a breakpoint is to dump specified information to the log file `ESTELLE.LOG`.

After maturing, a breakpoint may be *reset* or *suspended*. If it is reset, its pass count will be reset as well, so it will behave as though it had just been set by the user for the first time.

It is also possible to specify whether a breakpoint should return control to the user or simply continue with execution. The latter is useful when the breakpoint has an action such as reactivating a different breakpoint or dumping information. Often a breakpoint will have no action other than to return control to the user.

Breakpoints set on transitions and interaction points may be constrained to affect only the process that was active when they were set, or to hold across all peer processes with the same body type. In the latter case, any such process which satisfies the breakpoint conditions will cause the pass count to be decremented, and this pass count is shared between all such processes. In other words, a breakpoint with a pass count of five that is set on a single process will mature after that process has satisfied the breakpoint conditions five times, while one set on any process of that type will mature after it has been satisfied five times in total by any of the appropriate processes. Thus, breakpoints set on a single process give a finer degree of control than those set on process types.

When the *Set Breakpoint* command is selected from one of the window menus, the *Breakpoint Menu* is invoked (see Figure 2). The bottom few lines of this menu give details about the breakpoint, namely the *breakpoint number*, the item on which the breakpoint is being set, and the current active process, if relevant.

The *Action* fields specify whether the breakpoint should activate some other suspended breakpoint, and if so, which one (specified by its breakpoint number), and whether information should be dumped to the log file. The dumped information consists of any combination of transition and interaction point information, corresponding to the window contents and statistics tables. The information may be dumped for any of the following cases:

- The current process

Figure 2: The Breakpoint Menu

- The current process and all its peers of the same body type
- The current process and all its peers
- All processes

The information to be dumped is selected with the *Dump Pane* option, while the case is selected with the *Process* option.

The first few options in the *Breakpoint Menu* specify:

- *Pass Count* The number of times the breakpoint must mature before any action will be taken
- *Process* Whether the breakpoint applies to the current process only, or to all peer processes with the same body type (this is not applicable for line breakpoints)
- *Transition* If the breakpoint is set on a transition, is it for the enablement of the transition, or its execution?
- *Control* Should control return to the user when the breakpoint matures, or should execution continue?
- *Reset* Should the breakpoint be reactivated after maturing, or should it be suspended?
- *Active* Is the breakpoint currently active or suspended?

Once all the relevant information has been selected, the *Ready* command at the top of the menu should be executed. If you wish to cancel setting the breakpoint before selecting *Ready*, press ESC.

If the *View Breakpoints* command is selected from a window menu, the *Breakpoint Browser* is invoked. This browser allows the user to examine, modify or delete breakpoints. Suspended breakpoints have parentheses about their number. By pressing *Del* the user may delete the currently highlighted or all breakpoints; by pressing *Enter* the *Breakpoint Menu* is reinvoked for that breakpoint so that it may be edited.

11 The Log File

At the end of execution, the logfile contains a summary of queue traffic and transition firing statistics as well as information dumped as a result of breakpoint actions, and having the Log option on. This latter option is toggled using the *F8* key and turns full information dumping on for all processes at every scheduler iteraction. This allows a detailed trace of the execution of a specification to be captured. Be warned that such traces grow very large very quickly!

The log file enables us to determine the probabilities of transitions firing, and thus to easily calculate throughput and error rates of protocols.

The log file also contains matrices of *transition sequence execution counts* and *transition sequence delays*. These are the number of times each transition in each process was followed by each other transition in that process, and the mean delay in each case. These figures are used for building Markov process models of the long-term behaviour of the executing specifications when viewed as stochastic processes. At the end of the log, global matrices are printed out. The global transition sequence count matrix includes *linkage counts* between processes showing the relationships between transitions that `OUTPUT` interactions and the corresponding receiving `WHEN` transitions.

12 Technical Reference

This section details the limitations of the *PEW*, and its error messages. For details on the restrictions and extensions of this implementation of Estelle, refer to the Appendix.

12.1 System Limits

Hardware Requirements:

- 8086 processor running MS-DOS or PC-DOS (i.e., IBM PC, PC-AT or compatible), with at least 256kB of memory
- An EGA/VGA or compatible display is preferable.

Editor Limits:

Maximum line length	255 characters
Maximum character search set	15 characters
Maximum number of place marks	10
Maximum keystroke macro length	128 keystrokes

Compiler Limits:

Maximum string constant length	80 characters
Maximum number of case indices	256; all case values must fall in the range of 0..255.
Maximum allowed distinct identifiers	500
Symbol Table Size	30 kB
String Table Size	5000 bytes
Code Buffer Size	10000 words
Maximum identifiers in `EXIST` expression	16 (this also applies to the `ALL` and `FORONE` constructs).

Maximum Set Size	64; all set values must fall in range 0..63.
Maximum Scope Levels	16. `WITH` statements and others that introduce new identifiers all use one or more scope levels.

Interpreter Limits:

Maximum per-process heap size	1024 words (characters and integers each use up one word).
Maximum number of breakpoints	32

12.2 System and Editor Error Messages

Memory allocation failure
: An attempt to allocate memory failed. This error often results in the *PEW* aborting. If you have been editing a file, this will be saved to a temporary file and a message to that effect will be printed.

Illegal argument passed to function *fn*
: This is an internal error. It is unlikely to ever occur.

Cannot save screen image
: You attempted to run a DOS shell but the *PEW* could not allocate sufficient memory to save the screen contents. In this case, it is very unlikely that a shell could be executed successfully anyway!

Cannot find COMMAND.COM
: You attempted to run a DOS shell but the *PEW* could not find your copy of `command.com`.

Insufficient memory
: An attempt to run a DOS shell failed due to lack of available memory.

Error while trying to execute COMMAND.COM: *n*
: An error occurred during an attempt to run a DOS shell. The number corresponds to the DOS error code for whatever error occurred.

Cannot open file: *fl*
: You attempted to load a nonexistent file into the editor.

Illegal input: integer value required
You executed some command which expected a number as input but found something else instead.

Illegal input: line number out of range
You attempted to execute a *Goto Line* command to a line which does not exist.

Illegal tabstop value
You entered an invalid tab stop value.

Cannot open help index file

Cannot open help text file
You are missing one or both of the *PEW* help files in your PEWPATH directory, or current directory if the PEWPATH environment variable is not being used. Check that you have these files, and that you are in the same directory as them, or they are in the directory specified by PEWPATH.

Cannot record macro when already busy recording
You attempted to begin recording a keystroke macro while one was already busy being recorded. Keystroke macros cannot be nested.

Cannot play back macro while recording
You attempted to play back a keystroke macro while busy recording one.

Cannot assign macro to this key
Keystroke macros can only be assigned to the keys *Alt-0* through *Alt-9*.

Macro exceeds maximum length (n keys)
You exceeded the keystroke macro buffer size (currently 128 keystrokes).

12.3 Compiler Error Messages

Ambiguous identifier
You declared an identifier that has already been declared.

Assignment incompatibility
You are attempting an assignment of incompatible types.

Bad comment
You have not terminated a comment; the end of the file was reached while still in the comment.

Incompatible Types
You are attempting to operate on types that are not compatible.

Redefinition of Function's Parameters
You have already declared the types of the function's parameters in an earlier forward declaration.

Invalid Integer
You have used an integer constant which is out of the range –32768..+32767.

Invalid Label Reference
You have a GOTO to an undefined or invalid label.

Invalid String
A non-printable character was reached before the end of the string. You may not have terminated the string properly.

Invalid Assignment to Function Variable
You cannot assign to the function variable at this scope level.

Set Element Out of Range
All set elements must have values in the range 0..63. This means, for example, that sets of alphabetic characters are not allowed.

Invalid Time Unit
Valid timescales are HOURS, MINUTES, SECONDS, MILLISECONDS and MICROSECONDS.

Missing Queue Discipline - No Default
You declared an interaction point without a queue discipline, and no default queue discipline has been specified.

Missing System Class
You declared a module with body transitions but no process or activity attribute.

Too many labels
You have exceeded the allowed number of labels.

Too Many Levels
You have exceeded the allowed number of scope levels.

Invalid Ordinal Type
An ordinal type was expected where some other type was used.

Invalid Pointer Type
A pointer type was expected where some other type was used.

Redefinition of Procedure's Parameters
You have already declared the types of the procedure's parameters in an earlier forward declaration.

Invalid Index Range
A value which was out of the legal range was found. You may be assigning an integer to a variable which is of some subrange or enumeration type.

Case Constant Repeated
You have used a case constant more than once within the same case statement.

Role Repeated in Role List
You have used the same role twice in a single role list.

Invalid Type for Selector
The case selector constant expression is of a different type to the case selector index expression.

Invalid Signed Constant
Only integers may be signed; you have used a unary minus with some other type.

State Definition Repeated
You have already had a state-definition part in the current set of declarations; you may not have two.

Invalid Type
You have used a type in an inappropriate place.

Undefined Identifier
You have referenced an identifier that has not previously been declared.

Invalid Assignment to Control Variable
You may not assign to loop control variables within loops.

Invalid Non-local Control Variable
Loop control variables must be local variables.

Invalid Control Variable
The variable you are using to control a loop is of an inappropriate type.

Label Reference Unresolved in Block
You have a GOTO to a label which is never defined.

Case index out of range
You have used a case constant expression which is out of the allowed range 0..255.

Duplicate clause type in transition
You have more than one clause of the same type associated with the same transition block.

ANY clause not implemented
The ANY clause is not implemented in Version 1.2 of the *PEW*.

ANY constants not implemented
ANY constants must be of simple types. The PEW will compile a specification using ANY constants as thought the smallest non-negative allowed value for the ANY constant was specified, or if this is not possible, the largest negative value.

... types not implemented
The *PEW* only allows complete specifications that are executable; ... types are thus not allowed.

Dispose with tags not implemented
Only the basic form of `DISPOSE` is allowed; the variant form is not implemented.

EXTERNAL not implemented
The *PEW* only allows complete specifications that are executable; `EXTERNAL` functions and procedures are thus not allowed.

Functional parameters not implemented
Functional parameters have not been implemented in Version 1.0 of the *PEW*.

NEW with tags not implemented
Only the basic form of `NEW` is allowed; the variant form is not implemented.

Packed types not implemented
The *PEW* does not support type packing.

PRIMITIVE not implemented
The *PEW* only allows complete specifications that are executable; `PRIMITIVE` functions and procedures are thus not allowed.

Procedural parameters not implemented
Procedural parameters have not been implemented in Version 1.0 of the *PEW*.

Record type expected in WITH statement
The identifier following `WITH` is not a record.

Estelle construct illegal in procedure/function
You have used an Estelle construct that is prohibited within Pascal functions and procedures.

GOTO illegal in current context
GOTOs are only allowed within procedures and functions.

Illegal module variable reference
You have used a module variable in a context where it may not be used.

Identifier expected
The compiler expected an identifier but got something else.

Integer expected
The compiler expected an integer but got something else.

TO or DOWNTO expected
The keyword `TO` or `DOWNTO` was expected as the compiler is processing a `FOR` loop.

Bad factor in expression
You have an illegitimate identifier reference in an expression.

Bad constant
A constant was expected but not found.

Bad formal parameter syntax
A syntax error in a formal parameter list occurred.

Clause or BEGIN expected
The compiler expected a transition but got something else.

Syntax error
Some kind of syntax error occurred.

Incompatible roles in CONNECT or ATTACH
To `ATTACH` two interaction points, they must have the same roles. To `CONNECT` two interaction points, they must have opposite roles. You violated one of these rules.

Syntax error: ';' expected

Syntax error: ':' expected

Syntax error: ')' expected

Syntax error: '(' expected

Syntax error: comma expected

Syntax error: END expected

Syntax error: DO expected

Syntax error: '[' expected

Syntax error: '.' expected

Syntax error: ' ' expected]

Syntax error: OF expected
These are all self-explanatory errors, and are special cases of the general 'Syntax Error' error.

Out of symbol table space
The specification you are attempting to compile has exhausted the symbol table space. Try to reduce the number of declarations you have. Eliminate type declarations that are only used once, and add type declaractions where these are implicit and used several times.

String store overflow
You have exhausted the string store for storing identifier names. Try to shorten the names of identifiers to save space.

Out of code space
You have exhausted the size of the code buffer.

Too many lines in source program
Version 1.x of the PEW allows a maximum of 5000 source lines in a single specification.

Invalid type for exported variable
You have declared an exported variable which is not a Pascal variable type.

Expected an exported variable reference
An exported variable reference was found by the compiler, but the variable referenced is not an exported variable.

Expected module type or ordinal type in EXIST, FORONE or ALL
An inappropriate identifier was used in one of these constructs.

Only a single module variable allowed in EXIST, FORONE or ALL
Module domains for these constructs may have only a single module variable.

Illegal mix of module and ordinal types in EXIST, FORONE or ALL
These constructs allow both module and ordinal types, but not simultaneously.

Too many identifiers in domain list in EXIST, ALL or FORONE
You have exceeded the maximum allowed length of a domain list.

Illegal operation for pointer types
You have attempted an operation on pointer types that is not permitted.

Illegal clause type in initialisation transition
Initialisation transitions may only have `TO` and `PROVIDED` clauses.

Maximum number (*val*) of unique named transitions exceeded
The PEW keeps track of the firing frequencies of all named transitions for use in the ANALYSIS menu's dependent expression evaluator. As space must be reserved for this data, there is a limit on how many named transitions are permitted in a specification.

Interaction *name* has no associated OUTPUT statement
This interaction is used in WHEN clause(s) but does not occur in any OUTPUT statements. The transition(s) that refer to it can thus never be enabled, and are redundant.

Interaction *name* has no associated WHEN clause
This interaction is used in OUTPUT statement(s) but is never used in any WHEN clauses. This is a potential deadlock situation due to the strict FIFO nature of Estelle's queues.

Interaction *name* is not referenced
This interaction is never used in any WHEN clause or OUTPUT statement.

State '*name*' is unreachable
This state never occurs in a TO clause or a FROM..TO SAME clause pair. Thus it is not possible for it to ever be reached. This is not an error, but the state is redundant.

State '*name*' is a deadlock state
If this state is reached, it can never be left, as there are no FROM clauses that are enabled in this state. The prescence of such a state indicates a design error in your specification.

Bad nesting of clauses
You have badly nested clauses in the current transition. Specifically, you have used a clause type that is already on the stack of clauses for this transition.

Line *n*: (E*m*) Interaction in OUTPUT may not have a priority

This interaction is not associated with a priority-queueing channel type.

Line *n*: (E*m*) OUTPUT interaction cannot have a propogation delay

This interaction is not associated with a FIFO-queueing channel type.

12.4 Interpreter/Debugger Error Messages

Cannot open log file
The *PEW* failed to open the file `ESTELLE.LOG`. You may have some other application running, or your disk may be full. If you are running from a floppy disk, you may have a write-protect sticker on your disk.

Expression value in CASE statement has no corresponding label
You have not dealt explicitly with each possible case for the case statement index expression type.

Attempt to dequeue nonexistent interaction
This is an internal error. It should never occur.

Attempt to output interaction to unconnected IP
: You have attempted to execute an `OUTPUT` statement to an unbound interaction point.

Attempt to execute illegal instruction (n) at code offset o
: This is an internal error. It can only occur if stack overflow occurred but was never detected. It is unlikely to ever happen.

Stack overflow
: A process's stack has exceeded its bounds. Process stacks are usually expanded dynamically as required, unless the process performs dynamic memory allocation using `NEW`, at which point the stack is expanded by 1000 words and then frozen in size.

Attempt to expand process stack failed
: As mentioned above, process stacks are expanded dynamically as required. If this fails, it is due to either low memory or the stack becoming too big ($\geq$32kB).

Index or value *val* out of range *val*... *val*
: A variable of some ordinal type has exceeded its allowed range of values.

Attempt to connect an already connected interaction point
: An interaction point that is already bound was referenced in a `CONNECT` statement.

Attempt to attach a bound interaction point
: An interaction point that is already bound was referenced in a `ATTACH` statement.

Too many failed WHEN clauses in process
: As the *PEW* checks for interactions that cannot be dealt with, it maintains data structures for this purpose. You are attempting to execute a specification which exceeds the available data structure space for this purpose.

NEW failed to allocate heap memory
: An attempt to dynamically allocate memory failed because the request was too large or the stack is close to overflowing.

Attempt to DISPOSE illegal memory block
: An attempt to dispose a memory block that is not a currently valid memory block allocated by `NEW` was made. It is likely that you have never done a `NEW` on this pointer before calling `DISPOSE`.

No breakpoint corresponding to this number
: A reference to a non-existent breakpoint was made.

Division by zero
: The next divide instruction to be executed by the PEW has a zero-valued denominator.

12.5 Memory Allocation Failure Messages

Failed to allocate interaction

Failed to allocate a module entry

Failed to allocate a transition sequence table

Failed to allocate a module stack

Failed to allocate a module variable table

Failed to allocate an IP table

Failed to allocate a common queue

Failed to allocate a reception queue

Failed to allocate a transition table

Line n: (Em) Failed to allocate symbol table memory block

Line n: (Em) Failed to allocate compiler code buffer

Line n: (Em) Too many transitions in process

The number of transitions in a process is limited to 64.

A The *PEW* Estelle Implementation

This section gives a short summary of how the implementation of Estelle in the *PEW* differs from the Draft International Standard.

The following restrictions apply to the *PEW*:

1. `REAL` and `PACKED` data types are not supported.
2. The `ANY` clause is not supported.
3. The `NEW` and `DISPOSE` standard procedures do not allow the variant forms.
4. Partial specifications are not supported; i.e., `PRIMITIVE` and `EXTERNAL` functions and procedures and `ANY` types.
5. Procedures and functions may not be passed as parameters.

In addition, the compiler performs no purity checking of procedures and functions. The keyword `PURE` is ignored, and it is assumed that the user takes responsibility for purity checking.

The restriction in the standard of `WHEN` and `DELAY` clauses being mutually exclusive is not enforced by the *PEW* compiler. Instead, the combination is treated as follows: the `WHEN` clause must be continuously satisfied for the duration of the delay for the transition to be offered for execution. If at any stage during the delay the `WHEN` clause is no longer satisfied, then the delay timer associated with the transition is reset.

Additions to the *PEW* include:

1. `READ`, `WRITE` and `WRITELN` procedures for integer, character and string variables and constants. Fieldwidths are not supported. All I/O is directed to a pop-up window which can be viewed at any time in the interpreter by pressing *F7*. This can be overridden using a compiler directive as follows:

   - {$o0} – disable all output
   - {$o1} – send output to log file only
   - {$o2} – send output to window only (default)
   - {$o3} – send output to both log file *and* window

   It should be noted that this compiler directive generates an e-code instruction at the point where it occurs, so placement is important. The advantage of generating an instruction is that different WRITE statements can have different destinations. Once such an instruction is processed, all WRITE statements will be subject to it until the next such instruction is processed.

   It should also be noted that when logging (*F8*) is turned on, then the default is to send output to both the log file and the window.

2. A `RANDOM` function has been added to generate random numbers. This takes a single positive integer argument n and returns a pseudo-random number in the range $0..n–1$.

3. A `GLOBALTIME` function has been added which returns the current simulation time.

4. A `FIRECOUNT` function has been added which returns the number of times the containing transition has been executed.

5. A `QLENGTH` function has been added which returns the length of the reception queue of an interaction point. The interaction point must be specified as an argument to the function.

6. `OUTPUT` statements may be preceded by a compiler directive to specify their reliability. For example {`$R95`} `OUTPUT...` specifies an `OUTPUT` statement that is 95% reliable. This reliability applies only for a single `OUTPUT` statement, after which it is reset to 100%.

7. The *PEW* detects interactions at the head of IP queues that cannot be dealt with and discards these. Each time this occurs, a message is written to the log file detailing the occurence. This helps to check whether all possible interactions have been catered for.

8. The *PEW* compiler detects unused and unreachable states, interactions that can be sent but never received, or transitions that are only enabled upon reception of interactions that are never sent. Thus a reasonable degree of static validation is performed.

9. The *PEW* compiler allows the use of `ANY` constants. It uses the smallest possible positive value for such a constant, if possible; otherwise the largest value is used. Thus, ANY 10..20 will be treated as 10, ANY -10..10 will be treated as zero, and ANY -20..-10 will be treated as -10.

10. The *PEW* allows uniform, exponential, geometric and Poisson delay distributions. Selecting between these is discussed in Section 9.

11. Normally, a `WRITE` or `WRITELN` statement directed at the pop-up I/O window results in a short pause for you to examine the result before execution proceeds. This can be disabled and enabled at different points by using the *fast write* compiler directives {`$F--`} and {`$F+`}.

12. The execution log contains transition sequence count and transition sequence delay matrices for each process. In some cases, we may wish to restrict attention to a subset of the transitions in a process. This can be achieved to a limited extent by use of the {`$T`} directive, which may be used once per module, and divides the module's transitions up into two separate groups.

13. In Estelle, interactions are queued in FIFO order. This can be overridden in the *PEW* by use of the {`$QR`} (random queueing) and {`$QP`} (priority queueing) directives. These directives affect the next channel type declaration, after which FIFO queueing once again becomes the default. When declaring a channel type, priorities are assigned sequentially to each set of interactions declared, starting with zero (the highest priority). The current priority value can be set to a specific value by using the {`$S`*value*} directive. An example will help to illustrate this:

```
{$QP Select priority queueing}

CHANNEL prot_prov_chan(prot_role, prov_role);
    BY prot_role, prov_role:
        {** REJ frames have highest priority **}
        REJ(sv:seq_type; rv:seq_type);
        {** RR frames have priority 1 **}
        RR(sv:seq_type; rv:seq_type);
        {** RNR has same priority as RR **}
        {$S1} RNR(sv:seq_type; rv:seq_type);
        { I frames have priority 2 which is the lowest }
        I(sv:seq_type; rv:seq_type; data:data_type;crc:crc_type);
```

14. A propogation delay can be associated with an interaction point by preceding the IP declaration with the directive {`$D`*value*}. Interactions output through the IP will be timestamped, and may only be dequeued when the global time exceeds their arrival time. This facility is only supported for IPs that use FIFO queueing.

B An Overview of Estelle

B.1 Introduction

Estelle is a *formal description technique* (FDT) based on an extended state transition model (Estelle is a loose acronym for *Extended State Transition Language*). It is being developed as an International Standards Organisation standard by ISO/TC 97/SC 21, the subcommittee for Information Retrieval, Transfer and Management for the Open Systems Interconnection (OSI).

Estelle can be used to describe distributed, concurrent information processing systems. It is aimed in particular at describing the service definitions and protocol specifications of OSI layers. In the OSI model, each layer is separated from the layers above and below, and communicates with these via service requests. Apart from this communication, the various layers can be regarded as separate asynchronous systems. Furthermore, the peer layers on separate systems behave relatively asynchronously with respect to one another. A further characteristic of such systems is nondeterminism. Through a hierarchical structure of processes, Estelle provides facilities for both asynchronous parallel execution and nondeterministic sequential execution.

Essentially, Estelle is a subset of the programming language Pascal (ISO 7185) with extensions for concurrency based on an extended finite-state model. An Estelle program (called a **specification**) specifies a number of processes which may execute either sequentially or concurrently, or a combination of both. Each process is an extended finite- state automaton, consisting of a number of *states*, a set of possible *state transitions*, the *conditions* which must be satisfied for the transitions to be *enabled*, and the *actions* associated with each transition. The automata communicate with one another using *message passing* on FIFO queues. Each automaton typically represents a layer or part thereof of a protocol executing on some specific machine. A 3-layer protocol executing on two separate machines would thus typically be represented in Estelle using 3*2=6 automata. The entire system is dynamic and nondeterministic; new processes or automata may be freely created and destroyed during the execution of a specification.

This informal introduction to the syntax and semantics of Estelle assumes the reader is familiar with the Pascal language; only the changes and extensions to Pascal are considered.

B.2 Conventions and Outline

Several conventions have been used throughout this document. Within syntactic grammar rules, bold type has been used to represent terminals, italic type for nonterminals, and normal type for the grammar metalanguage. The metalanguage consists of the symbols:

=	Separates right and left hand sides of productions
|	Separates alternatives
[]	Indicates optional elements

{}	Indicates zero or more elements
+{}	Indicates one or more elements
()	Used for grouping purposes
<>	Used to identify nonterminals
...	Used to abbreviate a scalar range

Unconventionally, productions are not terminated with a period as this may cause confusion with punctuation symbols that are intended as language terminals.

Within the productions, upper case nonterminals represent Pascal constructs. In general, these productions are not explicitly given, but should be fairly obvious to the reader who is familiar with Pascal. Lower case nonterminals are used exclusively for Estelle constructs.

Within the text, important terms have been indicated in italic, while grammar terminals are once again indicated in bold.

To clarify the purpose of identifiers, they are often prefixed with an abbreviation to indicate their use. For example, *role-IDENT* is an identifier whose purpose it is to identify a role. These prefixes are included for semantic clarification only; all identifiers ultimately have the same production rule.

The various parts of a programming language are usually strongly interconnected; it is difficult to discuss any one aspect of the language in isolation from others. The breakdown used here is necessarily artificial. In an attempt to reduce forward references different parts of the language will be introduced as they are used; however, the detailed discussion of any particular part will be confined to the relevant section.

The first section is a brief overview of the additions to ISO Pascal. None of the additions are discussed in detail; they are dealt with individually in the remaining sections. Section B.4 considers the restrictions to ISO Pascal that have been introduced.

Section B.5 introduces some important terminology, while Section B.6 is a discussion of how finite-state automata may be realised using Estelle. It concentrates primarily on the external behaviour of the automata, regarding each automaton as a process in a multiprocess system. Section B.7 discusses the internals of the automata, namely, the specification of states and state transitions, and how these are interpreted during execution. Section B.8 concentrates on the communication between automata, that is, the message passing and shared variable facilities of Estelle.

As each new aspect of the language is covered, the syntactic production rules for that aspect are given, and the semantics discussed. A collected syntax is included in the appendix. Section B.2 describes the metalanguage used to specify the syntactic productions, as well as other conventions used.

B.3 Additions to ISO Pascal

The main additions to standard Pascal are the provision for concurrent systems and the provision for specifying finite-state automata. Other additions include:

- identifier names may have underscores in any position; this is the only difference from standard Pascal identifiers. The general production rule for identifiers is thus:

  ```
  <IDENT>   =  <letter> { <letter> | <digit> }
  <letter>  =  a | b | c |... | z | A | B | C |... | Z | _
  <digit>   =  0 | 1 | 2 |... | 9
  ```

- partial specifications are possible, allowing early syntactic checking, separate compilation or the specification of protocols without specifying irrelevant details. This is achieved by allowing constants to be declared with the value **any** and variables to be declared with the type .... In addition to functions and procedures being able to be forward-declared by qualifying them with the keyword **forward**, the **primitive** and **external** keywords can be similarly used to declare functions or procedures that are not defined within the particular file containing these definitions. An **external** routine must be declared within another file which is part of the specification; a **primitive** routine is one which is provided by the language implementor (typically a library function/procedure).

  This facility allows incomplete specifications to be compiled and thus checked for syntactic errors, access to external libraries of functions, as well as the ability to omit irrelevant parts of the protocol specification (for example, the type of a protocol data unit is not needed to define the protocol itself).

- the **all** repetitive statement has been added to the existing **repeat**, **while** and **for** statements.

- the structured statement **forone** has been added

- the expression factor **exist** has been added

- the simple statements **attach**, **detach**, **connect**, **disconnect**, **output**, **init**, **release** and **terminate** have been added.

The **init**, **release**, **terminate**, **all** and **forone** statements and **exist** expression are discussed in Section B.6; the **attach**, **detach**, **connect**, **disconnect**, and **output** statements are discussed in Section B.8.

B.4 Restrictions to ISO Pascal

The fact that Estelle supports concurrency has repercussions within the Pascal subset, primarily to allow functions and procedures to be reentrant. Several restrictions to procedures and functions have thus been introduced. They may not reference non-Pascal objects, although the Estelle statements **all**, **forone** and **exist** may still be used on the usual Pascal ordinal types. Functions must all be pure; that is, side-effects are prohibited. This is enforced through several restrictions: parameters

may be passed by value only and may not contain any pointers, references to global variables are prohibited and so are calls to any other non-pure routines. Procedures need not be pure; if they are, they may be declared as **pure procedure**, and may then be called from functions and other pure procedures.

All file operations have been removed, including the file type and predefined **text** type. Thus the execution of an Estelle specification is only meaningful in a controlled environment, or if the implementor adds his or her own **primitive** library routines for input/output.

Labels and **goto**s are also restricted - they may only occur within procedures and functions, and labels may only be associated with the end of the containing procedure or function; in other words, a **goto** acts similar to a **return** statement in a language such as 'C'.

B.5 Terminology

Several terms are used within Estelle to describe processes and interprocess communication. Although these terms are discussed within the relevant sections, it is appropriate to introduce some here.

A *channel* is a bidirectional FIFO queue used for interprocess communication. Each channel is identified by a *channel identifier*. The traffic over a channel (which consists of messages called *interactions*) can be different for each of the two directions; each direction is associated with a *channel role* which specifies what interactions may be sent in that direction. The endpoints of a channel are called *interaction points*.

An executing protocol *specification* in Estelle consists of one or more *modules* in a hierachical structure. These modules may have different *classes*: *activities* allow nondeterministic sequential execution (ie, if the scheduler must choose between a number of enabled activities, it will choose only one, and the choice is nondeterministic), while *processes* allow asynchronous parallel execution (ie, the scheduler will execute all of the enabled processes). This description is an oversimplification of what actually occurs, but is sufficient for now.

Each module comprises two parts: the *module header* specifies the external visibility of the module, while the *module body* specifies its internal behaviour.

The conditions associated with transitions are called *clauses*, and may be of several different types.

B.6 Automata

In this section, we will consider automata as processes in a multiprocessing system. We are thus primarily concerned with the creation and destruction of automata, not with their internal behaviour or intercommunication, which are discussed in sections B.7 and B.8 respectively.

At the top level, an Estelle program is a <*specification*>, identified by a name <*spec-IDENT*> and a class <*system-class*>, with an internal structure described by a <*body-defn*>. Queues for interprocess communication may be *individual* or *shared*;

usually such *queue disciplines* will be explicitly declared within queue declarations, although a global default can be specified with the optional <*defaults*>. This default must be provided if there are any queues declared without explicit queue disciplines.

Timescales are semantically meaningless and are simply used to express the intentions of the implementor. Time in the Estelle model need only be treated consistently; the actual units are irrelevant (except in real implementations), and serve only to express the intentions of the implementor.

<*specification*>= **specification** <*spec-IDENT*> [<*system-class*>]**;**
[<*defaults*>]
[<*time-options*>]
<*body-defn*>
end .

<*system-class*>= **systemprocess** | **systemactivity**

<*defaults*> = **default** <*q-discipline*> **;**

<*q-discipline*> = **common queue** | **individual queue**

<*time-options*>= **timescale** <*ts-IDENT*> **;**

<*ts-IDENT*> = **hours** | **minutes** | **seconds** |
milliseconds | **microseconds**

<*body-defn*> = <*decl-part*>
<*init-part*>
<*trans-decls*>

The <*body-defn*> of a specification consists of three parts: the *declarations*, an *initialisation part*, and a *transition declaration part*. The initialisation and transition declaration parts both specify groups of transitions (see section B.7 for details). Initially it is assumed that the specification exists in an initial state that results from executing a transition from its <*init- part*> if this is non-empty; otherwise it is in a *pre-initial state*.

<*decl-part*> = <*decls*>

<*decls*> = <*CONSTANT-DEFN-PART*>
| <*TYPE-DEFN-PART*>
| <*channel-defn*>
| <*module-header-defn*>
| <*module-body-defn*>
| <*ip-decls*>
| <*mod-var-decls*>

| <*VARIABLE-DECL-PART*>
| <*state-defns*>
| <*state-set-defns*>
| <*PROCEDURE-AND-FUNCTION-DECL-PART*>

<*init-part*> = { **initialise** <*trans-group*> }

<*trans-decls*> = { <*trans-decl*> }

<*trans-decl*> = **trans** <*trans-group*>

<*trans-group*> = +{ <*clause-group*><*trans-block*> **;** }

The declarations may be in any order, and, with the exception of the <*state-defn-part*>, any declaration or definition part may occur more than once. Apart from the usual Pascal objects, the declarations may contain *channels* (bidirectional FIFO message queues) and *interaction points* (queue endpoints), module *headers*, *bodies* and *variables*, and *states* and *state sets*.

An executing Estelle specification consists of a tree of *instances* of generic *modules*. The process structure may be dynamic; processes may at any time instantiate others with **init** statements and kill children processes with the **release** or **terminate** statements. These are discussed later in this section.

An Estelle module is described by its external visibility (which is defined by the <*modheader-defn*>) and internal behaviour (which is defined by the <*modbody-defn*>). These two components are separate, and thus modules with the same external visibility may exhibit different internal behaviour. Modules are bound to headers at compile time and bodies at instantiation time.

<*modheader-defn*> = **module** <*modhdr-IDENT*> [<*class*>]
[(<*param-list*>)] **;**
[**ip** +{ <*ip-decl*> **;** }]
[**export** +{ <*VARIABLE-DECL*> **;** }]
end ;

<*class*> = **systemprocess** | **systemactivity**
| **process** | **activity**

<*param-list*> = <*VALUE-PARAM-SPECIFICATION*>
{ **;** <*VALUE-PARAM-SPECIFICATION*> }

The only external access to a module instance is via its interaction points and *exported variables* which are shared with the instance's parent only. These exported variables are declared in the <*modheader-defn*>. The module header also defines parameters which are passed (by value only) to an instance of the module when the instance is created. The interaction points declared in the module header are

external interaction points; in other words, they may be accessed by the module's parent.

<*modbody-defn*>=**body** <*modbod-IDENT*> **for** <*modhdr-IDENT*> **;**
(<*body-defn*> **end ;**
| **external ;**)

Notice that two identifiers are used to identify the module body; the first is associated with the module body itself, and the second specifies which module header is being referenced. If the body definition contains the keyword **external**, then the actual body definition is in some other specification.

The relative behaviour of module instances in the hierarchy is determined by the nesting of their definitions as well as the class qualifiers **systemprocess**, **systemactivity**, **process** and **activity**. Processes allow for synchronous parallel execution, while activities allow nondeterministic sequential execution. A process may be created and released only by its parent. A module with a class qualifier is called *attributed*; the only non- attributed modules are inactive modules, which as their name suggests consist only of initialisation parts. Inactive modules may only parent system modules, which may in turn parent only process or activity modules. (System)process modules may contain either process or activity children, while (system)activity modules may contain only other activity modules.

A consequence of the above is that any initial state of a specification defines a fixed number of system instances and interaction point links above them; this structure once initialised cannot change since the parent modules of the systems are all inactive and hence cannot create any new processes after initialisation. Furthermore, as their parents are inactive and hence have no execution priority over the systems, the systems behave fully asynchronously with respect to one another.

Instances of modules are identified by *module variables*. References to exported variables and interaction points must be qualified by the name of the appropriate module variable. Arrays of module variables are allowed.

<*mod-var-decls*>=**modvar** +{ <*mod-var-decl*> **;** }

<*mod-var-decl*>= <*IDENT-LIST*> **:** <*modhdr-IDENT*>
| <*IDENT-LIST*> **: array** [<*index-type-list*>]
of <*modhdr-IDENT*>

The initial value of a module variable is undefined; an **init** statement will initialise the variable, and a **release** or **terminate** statement causes it to once again become undefined.

<*init-stmnt*> = **init** <*module-var*> **with** <*modbod-ident*>
[**(** <*EXPR-LIST*> **)**]

<*release-stmnt*> = **release** <*module-var*>

<*terminate-stmnt*> = **terminate** <*module-var*>

The execution of the **init** statement causes a new instance of the specified module body (and thus header) to be created, its parameters to be initialised with the expression values, and an initialisation transition from the module body (if one exists and is enabled) to be executed. After execution of the **init** statement, the module variable identifies the module instance.

The **release** statement allows a module to release children module instances. All external interaction points of the module instance are detached or disconnected, and the module instance and *all its descendants* are released and are no longer available. The **terminate** statement is a similar statement, except that it uses a form of detach known as *simple detach*. This will be described later.

The **all** statement is a repetitive statement which allows iteration over a normal ordinal type or over a set of module instances.

<all-stmnt> = **all** (*<domain-list>* | *<module-domain>*)
do *<STATEMENT>*

<domain-list> = *<IDENT-LIST>* **:** *<ORDINAL-TYPE>*
{ **;** *<IDENT-LIST>* **:** *<ORDINAL-TYPE>* }

<module-domain> = *<IDENT>* **:** *<modhdr-IDENT>*

The result of an **all** statement is the execution of the *<STATEMENT>* for either:

- all vectors of values of ordinal type(s) given in the *<domain-list>*; or
- all children module instances whose header definitions are identified by the *<modhdr-IDENT>* in the *<module- domain>*.

The order of execution is arbitrary. If the domain is empty, the statement is not executed. The bounds of a domain are evaluated once and are not affected by execution of the statement.

A similar statement is the **forone** statement:

<forone-stmnt> = **forone** (*<domain-list>* | *<module-domain>*)
suchthat *<BOOLEAN-EXPR>*
do *<STATEMENT>*
[**otherwise** *<STATEMENT>*]

This is much like the **all** statement, except that only one of the possible (vector of) values is used, and this must satisfy the Boolean expression.

To assist in these statements, the **exist** Boolean expression may be used:

<exist-one> = **exist** (*<domain-list>* | *<module-domain>*)
suchthat *<FACTOR>*

The semantics are obvious in the light of the **forone** statement: if an element of the domain exists that satisfies the *<FACTOR>*, then the expression returns TRUE, otherwise it returns FALSE.

B.7 States and State Transitions

We now consider the details of how Estelle allows state transitions to be specified.

Each instance of a module is a nondeterministic state transition system, characterised by a set of states, subset of initial states, and next-state relation. The initial states are defined by the module <*init-part*>, which describes the initial state, variable initialisation, and the initial hierarchy and interconnection structure of descendant module instances, if any. The one-to-many next-state relation is specified by transitions, each composed of a <*clause-group*> determining the pre- and post-states and any conditions which must be fulfilled for the transition to be *enabled*, and a <*trans-block*> defining the actions to be executed upon firing the transition. If the transition block is empty, the module is *inactive* and serves only as a parent to other modules.

<*init-part*> = { **initialise** <*trans-group*> }

<*trans-group*> = +{ <*clause-group*><*trans-block*> **;** }

All possible values of the control state of the EFSM must be enumerated in a state definition part:

<*state-defns*> = **state** <*IDENT-LIST*> **;**

States may be grouped together into state sets; this is simply a compact notation allowing a single *state set identifier* to reference a list of elements.

<*state-set-defns*> = **stateset** +{ <*state-set-defn*> **;** }

<*state-set-defn*> = <*state-set-IDENT*> **=** <*state-set-constant*>

<*state-set-constant*> = [<*state-IDENT*> { **,** <*state-IDENT*> }]

Transitions consist of <*clause-group*>s and <*trans-block*>s:

<*trans-decl-part*> = { <*trans-decl*> }

<*trans-decl*> = **trans** <*trans-group*>

<*trans-group*> = +{ <*clause-group*><*trans-block*> **;** }

<*clause-group*> = [<*provided-clause*>]
[<*from-clause*>]
[<*to-clause*>]
[<*any-clause*>]
[<*delay-clause*>]

[<*when-clause*>]
[<*priority-clause*>]

<*trans-block*> = <*CONSTANT-DEFN-PART*>
<*TYPE-DEFN-PART*>
<*VAR-DECL-PART*>
<*PROC-AND-FUNC-DECL-PART*>
[<*trans-name*>] <*STATEMENT-PART*>

<*trans-name*> = **name** <*IDENT*> **:**

The ordering of the clauses is not important. Transitions with **when** clauses are called *input transitions* as they are enabled only when some input condition occurs (i.e. some message is received); those without are *spontaneous transitions*. **when** and **delay** clauses are mutually exclusive. The optional <*trans-name*> is for documentation purposes only, and has no semantic effect.

Notice that the <*trans-decl-part*> consists of zero or more <*trans-decl*>s, each of which consists of a number of transitions preceded by the keyword **trans**. This grouping is important, and affects two aspects: the nesting of clauses, and the interpretation of **provided otherwise** clauses. We shall discuss the latter below. The nesting of clauses allows us to abbreviate transitions which have clauses in common. Clauses are held on a stack, which is affected in the following ways:

- upon entering a new <*trans-decl*>, the stack is cleared.
- upon entering a new <*clause-group*>, the stack is popped up to and including the clause type corresponding to the first clause in the <*clause-group*>. For example, if the new <*clause-group*> begins with a **provided** clause, the stack will be popped up to and including the **provided** clause on the stack, if any. If no such clause exists, the entire stack is popped.
- subsequent to this, all clauses in the new <*clause-group*> are pushed on to the stack, one by one.
- Finally, when the <*trans-block*> is entered, all clauses currently on the stack are applicable to that block.
- Once clauses are being pushed on to the stack, no duplicates are allowed. For example, if a **provided** clause is already stacked, the occurence of a **provided** clause is illegal.

We will now examine each type of clause in more detail.

<*provided-clause*> = **provided** (<*BOOLEAN-EXPR*> | **otherwise**)

The **provided** clause specifies a Boolean condition which must be met for the transition to be enabled. If it is omitted, it is assumed TRUE. **otherwise** is only permitted as the last clause of a transition group; it is TRUE is all other **provided** clauses in the transition group are FALSE.

```
<from-clause>        =  from <from-list>

<from-list>          =  <from-element> { , <from-element> }

<from-element>       =  <state-IDENT> | <state-set-IDENT>

<to-clause>          =  to <to-element>

<to-element>         =  same | <state-IDENT>
```

The *<from-list>* in a *<from-clause>* specifies those states from which a transition may be validly executed; if it is omitted, it is assumed satisfied. The *<to-element>* specifies the next state after the execution of the transition; if it is omitted or same then the state does not change.

```
<when-clause>        =  when <when-ip-ref> . <interaction-IDENT>
                            [ <interaction-arg-list> ]

<when-ip-ref>        =  <ip-IDENT> [ [ <ip-index> { , <ip-index> } ] ]

<ip-index>           =  <CONSTANT> | <VAR-IDENT>

<interaction-arg-list> =  ( <interaction-arg-IDENT>
                            { , <interaction-arg-IDENT> } )
```

A **when** clause is satisfied if the interaction specified by the *<interaction-IDENT>* is at the head of the queue associated with the interaction point indicated by the *<when-ip-ref>*. The interaction at the head of the queue is only dequeued as part of the execution of the transition. The *<interaction-arg-list>* is optional; if it is included, it must be complete and in the same order as in the corresponding *<interaction-defn>*. Section B.8 discusses interactions in more detail.

```
<delay-clause>       =  delay ( (<EXPR> , <EXPR>
                            | <EXPR> , *
                            | <EXPR>)
                            )
```

Some transitions may contain **delay** clauses specifying a minimum and maximum delay time; provided these are interpreted consistently time in the Estelle model is considered implementation dependent. A **delay** clause

delay (E1,E2)

specifies that the transition should not be selected for execution before E1 consecutive time-units of it being enabled have elapsed. Once E1 has elapsed, the transition may or may not be selected for execution (due to nondeterminism), even if it is the only transition enabled. However, once E2 units have elapsed, if the transition is the only one enabled it must be selected. "**delay (E)**" is the same as "**delay (E,E)**". An asterisk for the second expression indicates that there is no upper bound, so it is possible the transition may never be selected for execution.

<priority-clause> = **priority** *<priority-constant>*

<priority-constant> = *<UNSIGNED-INTEGER>* | *<CONSTANT-IDENT>*

priority clauses are used to attach priorities to transitions. Note that priorities are only relevant amongst peer processes, as parent transitions always have priority over children transitions. If the **priority** clause is omitted, the lowest priority is assumed. Zero is the highest priority, after which they decrease systematically.

<any-clause> = **any** *<domain-list>* **do**

<domain-list> = *<IDENT-LIST>* **:** *<ORDINAL-TYPE>*
{ **;** *<IDENT-LIST>* **:** *<ORDINAL-TYPE>* }

any clauses provide a shorthand way of specifying a number of clauses. A transition with an any clause is equivalent to as many transitions as there are distinct values in the *<ORDINAL-TYPE>*, one for each possible value. The identifiers in an **any** clause will typically be array indexes used to refer to arrays of interaction points in **provided** clauses.

The EFSM represented by a module body has all its initial states defined in the initialisation part:

<init-part> = { **initialize** *<trans-group>* }

The initialisation transition group is executed only once when a module instance is created. Only one transition block is executed, even if several are enabled (the choice being nondeterministic). Only **to** and **provided** clauses are allowed in the initialisation part.

As was mentioned in the previous section, systems allow for synchronous parallel execution, while activities allow nondeterministic sequential execution. The fact that we have both concurrency and nondeterminism means that at any stage, several transitions may be offered for execution. A transition offered by a parent always has

priority over one offered by a child. If a parent offers no transition but its children do, we first determine, for each child, the transition(s) selected for execution. If the parent is a (system)process then all of these transitions are selected for execution; otherwise one child is selected (the choice is nondeterministic) and its transition(s) used. This may be made more precise as follows:

Consider the process hierarchy as a tree. The transitions selected for execution ($AS(gid_P)$) may be synthesised bottom up using the following definition for any particular non-leaf node P with children P_i:

- If P is active and offers a non-null transition, then this transition is selected (Parent priority).
- Otherwise, if P is an (system)activity, then $AS(gid_P)$ is one of the $AS(gid_{P_i})$ (the choice is nondeterministic).
- Otherwise, if P is a (system)process, then $AS(gid_P)$ is the union of all the $AS(gid_{P_i})$ (parallel execution).

The execution of a specification thus proceeds as follows: for each system, a set of transitions are selected for execution. Once selected, a transition must eventually execute. Once all of the selected transitions have been executed for all systems, a new set is selected for each system, and the process repeats.

State transitions are atomic; thus, for the execution of Estelle specifications on uniprocessing systems, a process scheduler may be invoked upon the completion of any state transition to select the next transition (and hence process) for execution.

B.8 Interprocess Communication

Interprocess communication in Estelle is by message or *interaction* passing on bidirectional FIFO message queues (*channels*) connected at their endpoints (*interaction points*). As the process hierarchy may be an arbitrary structure, the interconnections between processes may span several levels of the hierarchy. Estelle does not allow the direct connection of any one interaction point to any other interaction point; instead, a connection between two processes must be established by a common parent of both processes (which may in fact be one of the processes involved). The Estelle **attach** statement then allows for the extension of the connection down the process hierarchy as necessary. It is important to distinguish between **attach** and **connect**; **connect** is used to establish connections while **attach** is used to extend those connections down through the process hierarchy.

Each interaction point may be used to send interactions, and is also associated with a FIFO queue (which may be shared by other interaction points of the same module) from which it can receive interactions. The set of interactions which may validly be sent and received through an interaction point are determined by the interaction point's <*channel-defn*>.

<*channel-defn*> = <*chan-heading*><*chan-block*>

<chan-heading> = **channel** *<chan-IDENT>* (*<role-list>*) ;

<role-list> = *<role-IDENT>* , *<role-IDENT>*

<chan-block> = +{ *<interaction-group>* }

<interaction-group>= **by** *<role-IDENT>* [, *<role-IDENT>*] :
+{ *<interaction-defn>* }

<interaction-defn> = *<interaction-IDENT>*
[(*<VALUE-PARAM-SPECIFICATION>*
{ ; *<VALUE-PARAM-SPECIFICATION>* })] ;

As can be seen, an interaction can have a number of value parameters; none of these may be of a pointer-containing type. For a given interaction point, a module assumes a role declared by a *<role-IDENT>*. A module may send interactions associated with its assumed role; a module that assumes the opposite role may receive interactions associated with the first role.

An interaction point is an abstract bidirectional interface through which a module may send or receive interactions. Each interaction point has three attributes:

- the channel
- the role
- the queueing discipline

Any interaction associated with the role may be sent through the interaction point; any interaction associated with the opposite role may be received through the interaction point.

The queueing discipline determines whether the queue associated with that interaction point may be shared with other interaction points of that module instance or not (common or individual).

<ip-decls> = **ip** +{ *<ip-decl>* ; }

<ip-decl> = *<IDENT-LIST>* : *<ip-type>*
| *<IDENT-LIST>* : **array** [*<index-type-list>*]
of *<ip-type>*

<ip-type> = *<chan-IDENT>* (*<role-IDENT>*) [*<q-discipline>*]

<index-type-list> = *<INDEX-TYPE>* { , *<INDEX-TYPE>* }

Interaction points may be referenced only by the operations **connect**, **attach**, **disconnect**, **detach**, the **when** clause of a transition, and in the **output** statement. Internal interaction points, which are defined in the <*decl- part*> of a <*body-defn*>, may only be bound and unbound using **connect** and **disconnect**.

<*connect-stmnt*> = **connect** <*connect-ip-ref*> **to** <*connect-ip-ref*>

<*connect-ip-ref*> = <*internal-ip-ref*> | <*child-extern-ip-ref*>

<*disconnect-stmt*> = **disconnect** (<*connect-ip-ref*> | <*module-var*>)

<*attach-stmnt*> = **attach** <*external-ip-ref*> **to** <*child-extern-ip-ref*>

<*detach-stmnt*> = **detach** (<*external-ip-ref*> | <*child-extern-ip-ref*>)

<*internal-ip-ref*> = <*ip-ref*>

<*external-ip-ref*> = <*ip-ref*>

<*child-extern-ip-ref*> = <*module-variable*>.<*external-ip-ref*>

<*ip-ref*> = <*ip-IDENT*> [[<*INDEX-EXPR*>
{ , <*INDEX-EXPR*> }]]

The two interaction points referenced must be declared with identical channel identifiers, and opposite role identifiers for **connect** or identical role identifiers for attach. After execution of the **connect** statement, any interactions output through an interaction point are received at the other (or its descendants if that interaction point is attached). A **connect** statement may connect internal interaction points and child external interaction points together in any combination.

The **disconnect** statement causes the specified interaction point, or all connected external interaction points of the specified child module instance to be unbound. The interactions for the interaction point(s) still remain in the queue, and may still be processed.

The **attach** statement attaches two interaction points, the first of which may be currently bound and the second of which may later be bound by an action of the child. An interaction point at the end of a sequence of bound interaction points is a *connection endpoint*. **output** statements at connection endpoints always cause their interactions to be queued at the interaction point bound to the opposite connection endpoint.

When the **attach** statement is executed, interactions present in the external interaction point queue which came through this queue are removed, and appended to the queue of the external interaction point of the lowest level descendant module instance which attached to the child external interaction point. The interactions

for two interaction points whose queuing option is individual queue within a parent module may be combined into a common queue of a child through **attach** operations.

The **detach** statement causes the specified interaction point and the one (and only one) to which it has been attached to be unbound. Any interactions queued at a connection endpoint at a lower level than the current level which were sent via the specified interaction point are appended onto the interaction point's queue before the interaction points are detached. A variant of this is called simple detach; in this case no interactions are moved. The terminate statement uses simple detaches rather than the conventional detach.

<output-stmt> = **output** *<interaction-ref>*[*<ACTUAL-PARAM-LIST>*]

<interaction-ref>= *<ip-ref>* **.** *<interaction-IDENT>*

The **output** statement results in the interaction (plus its parameters, if any) to be appended onto the queue assigned to the (other) connection endpoint. Outputs made through an external interaction point may be observed in the target queue only after the whole issuing transition is completed, due to the atomicity of transitions.

Interactions are received and removed from queues by executing transitions with **when** clauses. It may be necessary to examine a queue without dequeuing an interaction, if the transition containing the **when** clause also contains a **provided** clause which refers to one or more of the interaction paramaters. In this case, the interaction at the head of the queue (assuming the **when** clause is satisfied) must first be examined to ensure that the provided clause is also satisfied, before the interaction can actually be dequeued.

æ

C Evaluation and Bug Report

As the *PEW* is still under development, it would be appreciated if you could complete this form and return it to Graham Wheeler, DNA Laboratory, at the address on the front of this manual. All constructive criticism is welcomed.

I liked the following features of the *PEW*:

I disliked the following features of the *PEW*:

I would like to see the following features added:

I experienced the following bugs (please specify if the bug is repeatable, what circumstances caused it, what the symptoms where, and how fatal it was):