

# ECL 3.0 Documentation

Luciano Lavagno, Roberto Passerone, Ellen Sentovich  
Cadence Berkeley Laboratories  
2001 Addison Street, 3<sup>rd</sup> floor  
Berkeley, CA 94704-1103

May 3, 2008

(c) Copyright 2001 Cadence Design Systems

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Installing and Running ECL</b>	<b>3</b>
2.1	Required Software . . . . .	3
2.1.1	Java . . . . .	3
2.1.2	Esterel . . . . .	3
2.2	Other related software . . . . .	3
2.2.1	C compiler . . . . .	4
2.3	Obtaining and Compiling ECL . . . . .	4
2.4	Running Examples . . . . .	5
2.4.1	An Esterel Simulation Example . . . . .	5
2.4.2	An Example with User-Defined Types . . . . .	5
<b>3</b>	<b>ECL Command-line Interface</b>	<b>6</b>
3.1	ECL Compilation Scenarios . . . . .	6
3.2	ECL Compiler Options . . . . .	7
<b>4</b>	<b>ECL Source-level Debugging</b>	<b>8</b>
<b>5</b>	<b>Using XECL</b>	<b>9</b>
<b>6</b>	<b>ECL Syntax</b>	<b>10</b>
6.1	File Naming . . . . .	10
6.2	Modules and their Interfaces . . . . .	10
6.3	Statements . . . . .	11
6.4	Data Expressions . . . . .	12
6.5	Signal Expressions . . . . .	13
6.6	Always-true Signal Expressions . . . . .	13
6.7	Delay Expression . . . . .	13
6.8	Module instantiation . . . . .	13
<b>7</b>	<b>Notes on Semantics</b>	<b>14</b>
7.1	Module Execution . . . . .	14
7.2	Halting . . . . .	15
7.3	Signal Waiting . . . . .	15
7.4	Signal Emission . . . . .	15
7.5	Signal Test . . . . .	15

7.6	Expressions . . . . .	15
7.7	Preemption . . . . .	15
7.8	Concurrency . . . . .	16
7.9	Looping . . . . .	16
<b>8</b>	<b>Miscellaneous Topics</b>	<b>16</b>
8.1	Code Splitting . . . . .	16
<b>9</b>	<b>Application Notes</b>	<b>17</b>
9.1	Common Problems . . . . .	17
9.2	On Causality and Loops . . . . .	18
9.3	On Causality and Data/State . . . . .	18
9.4	User-defined Types . . . . .	21
9.5	Initializing Variables . . . . .	21
9.6	Using Constants . . . . .	22
9.7	Empty Modules . . . . .	22
9.8	Module Arguments . . . . .	22
9.9	Debugging Reactivity . . . . .	23
9.10	On Esterel Traps . . . . .	23
9.11	ESTEREL Keywords . . . . .	23
<b>10</b>	<b>Acknowledgements</b>	<b>24</b>

# 1 Introduction

This document describes the 3.0 release of ECL. ECL is both a language and a compiler. The language is basically C with ESTEREL-inspired reactive constructs added. The compiler parses ECL and splits it into pure C and pure ESTEREL parts. The remaining ECL flow may use a variety of compilers and synthesis tools to create a mixed hardware/software implementation. See [1] for more information about ECL.

The ECL compiler is released to the public, and we encourage its wide use and feedback from the users. All users should read the license, which is contained in the top-level directory of the release. More information can be obtained at <http://ecl.sourceforge.net>. Feedback should be sent to [ecl-devel@lists.sourceforge.net](mailto:ecl-devel@lists.sourceforge.net).

## 2 Installing and Running ECL

The ECL compiler has been developed and tested on Sun Solaris, Linux, and NT. The release contains source Java code for the ECL compiler, the ECL license, documentation, two tutorial examples, and a suite of regression tests.

### 2.1 Required Software

#### 2.1.1 Java

A Java compiler and a Java virtual machine are required to create the ECL compiler and compile ECL examples. The Sun Java Development Kit `jdk1.3` was used for the development of ECL, and it is available from the Sun web-site at <http://java.sun.com/jdk/index.html>. However, there are no known dependencies on a specific version of Java, and any compiler, virtual machine and real-time environment should be usable. After installing this software, one must set the environment variables `JAVAC` and `JAVAVM`, which point to the Java compiler and the Java virtual machine respectively. If a pathname is used when setting these variables, this may need to be changed for different environments. For example, `JAVAC=\java_path\bin\javac` will work only on NT, and `JAVAC=/java_path/bin/javac` will work only on UNIX (or cygwin), while `JAVAC=javac` will work on both provided the rest of the path is in the `PATH` environment variable.

#### 2.1.2 Esterel

The ESTEREL compiler is required to compile the output of the ECL compiler to a C model. ESTEREL `v5_91` was used in the development of ECL, and is available from <http://www.esterel.org>. The ECL compiler option `-ESTEREL` compiles an ECL file to an ESTEREL simulation target. For this option to be successful, the ESTEREL variable on NT (`ESTERELHOME` on Unix) should be set correctly to find `%ESTEREL%\lib\libcsimul.a`. The ESTEREL installation should correctly set this variable.

Note, however, that versions of the `libcsimul.a` library distributed with ESTEREL before `v5_90` do not have some features that are needed for correct ECL file simulation. Thus, the library contained in the these older ESTEREL distributions should be replaced with the ones contained in the ECL distribution, which are found in `ecl/lib/[arch]/libcsimul.a` (where `arch` is one of `sol2`, `linux` or `winnt`).

### 2.2 Other related software

VCC is a product of Cadence Design Systems and a license must be purchased to use it. See <http://www.cadence.com/systems>. Instructions for integrating ECL and VCC, as well as a tutorial example, are in Section ??.

### 2.2.1 C compiler

It is necessary to have a C compiler to run the C code generated by ECL and by ESTEREL. While on UNIX variants this is often bundled with the OS, on Windows NT you have to either purchase Microsoft Developer Studio, or download CYGWIN from <http://sourceware.cygnum.com/cygwin/>.

After installing this software, one must set the environment variables `CC`, which points to the C compiler (e.g., `cc` or `gcc` on UNIX, `cl` or `gcc` on NT).

The environment variable `OS` is also checked, if present, to force ECL to believe that it is running on a specific operating system. The default is to infer it from the Java runtime, but this may fail. For example, when running under Cygwin on NT, the OS is recognized as NT even though file naming convention, compiler name and so on should be set as under Linux. In this case setting `OS` to Linux should ensure that ECL behaves properly.

Setting environment variables such as `CC` can be done

- in `bash` by adding to the `$HOME/.bashrc` file the lines `CC=cl` and `export CC`,
- in `csh` by adding to the `$HOME/.cshrc` file the line `setenv CC cl`,
- on Windows NT by using the Environment tab of the System element of the Control Panel.

### 2.3 Obtaining and Compiling ECL

1. Download the latest release from <http://sourceforge.net/projects/ecl>. (Scroll down to Latest File Releases and select the most recent.)
2. Define the variables `JAVAC` and `JAVAVM` in the environment, which point to the Java compiler and the Java virtual machine respectively.
3. Define the variable `CC` in the environment, which points to the C compiler.
4. Define the variable `ESTEREL` on NT (on Unix it is `ESTERELHOME`), which is the root of the `ESTEREL` installation.
5. Create an ECL directory, for example `%ECL%`, and place `ecl.tar.gz` in this directory.
6. Add `%ECL%\ecl\src`, `%ECL%\ecl\util\JavaCup`, `%ECL%\ecl\util`, and `%ECL%\ecl\xecl` to your `CLASSPATH` environment variable.
7. Add `%ECL%\ecl\script` to your `PATH` environment variable.
8. `cd %ECL%; gunzip ecl.tar.gz; tar xf ecl.tar`
9. `cd %ECL%\ecl\util; make`
10. `cd %ECL%\ecl\src; make`
11. `cd %ECL%\ecl\xecl; make`
12. If you have a version of Esterel lower than 5.91, then `cd $ECL/ecl/lib/linux` or `cd $ECL/ecl/lib/sol2` or `cd %ECL%\ecl\lib\winnt` (depending on whether you are running ECL on a linux or Solaris or Windows NT operating system) and copy (use `copy` on NT and `cp` on Unix) `libcsimul.a` to `%ESTEREL%\lib` on NT and `$ESTERELHOME/lib` on Unix.

As a very simple test, run the following commands:

1. `cd %ECL%\ecl\test\abro`
2. `ecl abro.ecl`

## 2.4 Running Examples

### 2.4.1 An Esterel Simulation Example

The `ABRO` example is a standard in the `ESTEREL` example set. The behavior is to wait until at least one occurrence of `A` and one of `B` have occurred, then emit the output `O`. If at any time the signal `RESET` is detected, the behavior resets. In our version of this example, the signals `A`, `B` and `O` have integer values, and when `O` is emitted, its value is the sum of the values of `A` and `B`.

Execute the following commands to observe this behavior using the `ESTEREL` textual simulator (the output of the commands is shown here with the user-typed input):

```
cd %ECL%\ecl\test\abro
C:\ecl\ecl\test\abro> ecl -ESTEREL abro.ecl
C:\ecl\ecl\test\abro> abro.exe
abro> ;
--- Output:
abro> A(1);
--- Output:
abro> B(3);
--- Output:  O(4)
abro> A(2);
--- Output:
abro> RESET;
--- Output:
abro> B(4);
--- Output:
abro> A(10);
--- Output:  O(14)
etc...
```

Alternatively, one can simulate the ready-made input file, save the results, and compare these against a “golden” version:

```
abro.exe < abro.in > abro.test
fc abro.test abro.gd
```

The compilation and regression test simulation can be executed together automatically with the `-TEST` option:

```
ecl -TEST abro.ecl
```

### 2.4.2 An Example with User-Defined Types

The module `%ECL%\ecl\test\type_ex.ecl` contains a small example with a user-defined type. To run an Esterel simulation of this example, execute the following:

```
cd %ECL%\ecl\test\type_ex
ecl -ESTEREL type_ex.ecl
type_ex.exe -novarcheck
type_ex> ;
type_ex> a(3);
type_ex> b(6);
type_ex> c("6 7");
--- Output:  o(22) p(3 5) q
... etc ...
```

To read and write user-defined types in an Esterel simulation, the user must define the appropriate functions for converting between a string and the type. These functions are documented in the `ESTEREL` manuals, and examples are found in `type_ex.ecl`. These functions and any needed `extern` declarations can be placed directly in the `ECL` code, as they are for `type_ex.ecl`. However, they should be surrounded by `#ifdef STRL_SIMUL` since they may not cause problems when used with other programs (`VCC` for example) that do not allow modules to perform I/O operations on the console. This compilation flag is set when the `-ESTEREL` option is given to

the ECL compiler. The `-novarcheck` option instructs the ESTEREL simulator not to force all variables to be initialized. For more on variable initialization, see Section 9.5.

## 3 ECL Command-line Interface

This section contains two subsections: one which describes some typically scenarios for running ECL, and the other which details all the compiler options.

### 3.1 ECL Compilation Scenarios

There are three typical uses of the ECL compiler:

1. Compile the ECL file to an ESTEREL and a C file.
2. Compile the ECL file to an ESTEREL simulation model.
3. Compile the ECL file to a block that can be imported into a VCC behavioral diagram.

These flows are described below.

`ecl filename.ecl` Runs the ECL compiler on the file `filename.ecl` (in the current directory). This creates the ESTEREL and C files, `filename.str1` and `filename.h`, in the current directory.

`ecl -ESTEREL filename.ecl` Runs the ECL compiler on `filename.ecl`, runs the ESTEREL compiler on the resulting `filename.str1` file, and compiles and links the results to create a executable for simulation.<sup>1</sup> The simulation can be started by executing either `filename.exe -novarcheck` (which is linked with the `libcsimul.a` library), or by executing `xes filename`.<sup>2</sup>

`ecl -TEST filename.ecl` Similar to the `-ESTEREL` option, but also runs a regression test simulation on the result. It expects the files `filename.in` (the input stimulus file) and the file `filename.gd` (the correct response file) to be present in the current directory.

`ecl -VCC libname.cellname -DEFAULTTYPELIB typelibname filename.ecl` Intended to be used inside VCC; see Section ???. Runs the ECL compiler on `filename.ecl`, produces files for the VCC Whitebox C code wrapper, runs the ESTEREL compiler, and imports the result into the VCC workspace as cell `cellname` in library `libname`. The result is a module `libname.cellname:symbol` that is a block ready to be instantiated in a VCC behavioral diagram. The `-MAINMODULE` option may be specified to identify the main module if there are several in the ECL source file. If there are user-defined types, and the `-IMPORTTYPES` option is specified, these are imported into the library `typelibname`. Note that this overwrites any existing types in this library with the same name. To import a block with some types already defined, and others which need to be defined, and using several type libraries, see below and the documentation distributed with the add-on module for VCC users. Both `libname` and `typelibname` must be libraries in the VCC workspace. (They can be added to the workspace with the `Add Library` command in `Create`.) The `-NOIMPORT` option can also be used to update an existing ECL block in VCC without recreating the symbol and interface information.

`ecl -CLEAN filename` Cleans all of the ECL-generated files for module `filename`. To clean all ECL-generated files, on Unix execute `foreach f ( *.ecl ) ecl -clean $f end`, and on NT execute `for %f in (*.ecl) do ecl -clean %f`.

---

<sup>1</sup>Note that certain auxiliary functions must be defined by the user when they are user-defined types. See the ESTEREL manuals or the example in Section 2.4.2.

<sup>2</sup>This latter uses the `xes` library, which does not have the option to turn off the variable initialization check, and thus may not be simulatable if user-defined types are involved. We do not have a version of the `xes` library with this option.

## 3.2 ECL Compiler Options

The complete list of the ECL compiler options (lower case forms are also accepted) is given below.

1. Options defining the compiler mode (by default only the C preprocessor and the ECL compiler are executed):
  - ESTEREL** : Creates an ESTEREL-simulatable model.
  - CLEAN** : Cleans up all the temporary files generated by the ECL compiler.
  - VCC lib.cell** : Generates a C simulation model for VCC. Intended to be used inside VCC; see Section ??.
  - POLIS** : Generates a wrapper to import (as a black box, i.e., without estimation capabilities and for software implementation only) into POLIS.
2. General use options:
  - G** : Generate debug information so that by running a standard C debugger on the generated executable one can see the ECL source file, examine the variables, and so on, as discussed in Section 4 below.
  - P** : Use procedure calls from ESTEREL to C (default true). Creates procedure calls rather than function calls whenever possible.
  - F** : Use function calls from ESTEREL to C (turns off default procedure convention). Creates function calls rather than procedure calls whenever possible. This option is implied when importing an ECL module into POLIS, since POLIS does not support procedures.
  - C** : Prefer C translation of ECL statements that can be both Esterel and C (default).
  - E** : Prefer ESTEREL translation of ECL statements that can be both Esterel and C (turns off default C preference).
  - CHECK** : Do a simple ECL-usage check on the input file.
3. C preprocessor options:
  - D macro value** Defines a macro, with the same meaning as in the standard C preprocessor (that on UNIX would use the `-Dmacro=value` syntax).
  - I directory** Adds the directory to the include search path.
4. ESTEREL simulation options:
  - SF** : When an ECL file contains user-defined types, the ESTEREL compilation will expect certain type/text conversion functions and type comparison functions to be defined. The user should define these in the ECL source file if an ESTEREL simulation is to be run. If these functions have not been defined yet (for example, because one wants just compilation to complete), this flag will give dummy definitions to them.
5. VCC simulation options (intended to be used inside VCC; see Section ??):
  - VCC lib.cell** : Generates the Whitebox-C wrapper code and imports the module into VCC. The files `white.c`, which contains the interface for VCC, and `poindexterTypes.h`, which contains the information for importing user-defined types, are created. A main module must be determined for this operation to be successful; the `-MAINMODULE` option may be specified to identify the main module. The module is imported into the workspace (defined by the `-WORKSPACE` option) in library `lib` as cell `cell`. If there are user-defined types (`typedef` definitions), the type library should also be defined, in one of three forms:

- TYPELIB lib typename** : When used with the -VCC flag, specifies the library `lib` where the user-defined type (`typename` must be the `typedef` name) resides (or is imported if -IMPORTTYPES is specified).
- DEFAULTTYPELIB lib** : When used with the -VCC flag, specifies the library where all the user-defined types that are not assigned to a specific library by the first form reside.
- TYPELIBFILE filename** : Can be used in place of typing many -TYPELIB arguments. All the -TYPELIB arguments can be stored in `filename`. Thus, for example, the first line of `filename` may contain "mylib=mytype", and similarly for the rest of the lines.
- MAINMODULE name** : When used with the -VCC flag, specifies the name of the main module in the ECL file. If the user does not use this option, by default if there is only one module in the ECL file, it is used as the main module. Otherwise, the ECL compiler searches for a module with the same name as the base name of the input file. Otherwise, an error is reported to the user.
- WORKSPACE workspace** : When used with the -VCC flag, specifies the workspace directory where the `cds.lib` file is found.
- NOIMPORT** : When used with the -VCC flag, creates all the appropriate files for updating the VCC model (which has already been imported), and then simply copies these files to the library and cell name given to the -VCC flag. The -WORKSPACE argument must provide the full pathname of the current workspace. For updating a module, a full import is not needed (and in fact destroys symbol and interface information that may be used by connected modules).
- IMPORTTYPES** : When used with the -VCC flag, imports user-defined types (`typedefs`) into the libraries specified as shown above.

#### 6. POLIS simulation options:

- POLIS** : Generates an ESTEREL wrapper that can be compiled by the `strl2shift` POLIS pre-processor. The actual ESTEREL and C code generated by the ECL compiler are placed in two files with the suffix `_polis.strl` and `_polis.h` respectively, and the top-level ESTEREL compiler output is called by the wrapper.  
This elaborate scheme for importing into POLIS is due to the fact that POLIS software synthesis assumes side-effect-free function calls, that on the other hand are essential to the ECL compilation algorithm. See the `ecl_polis` script in the `script` sub-directory for an example of usage of ECL and POLIS.

#### 7. Compiler debugging options:

- TEST** : Creates an ESTEREL-simulatable model, runs this model on the file `filename.in` and compares the results with the file `filename.gd` (where `filename` is the name of the ECL module). `filename.in` and `filename.gd` must be in the current directory.
- PD** : ECL parser debug mode. DO NOT USE unless you want a huge output.
- CD** : ECL compiler debug mode. DO NOT USE unless you want a huge output.
- NG** : Disable grouping of statements (one procedure call is generated for each C statement in a block, rather than a single call for the block).

## 4 ECL Source-level Debugging

With the -G flag set, line number information about the source ECL file and debugging information about source variables and signals is written to the generated files. By using any C debugger (e.g. `gdb`, `dbx` or Microsoft Visual Studio) one can set a breakpoint on any ECL file

source line and print out the value of the source variables. One can also print out the presence information of signals, by using e.g. the name `E_sig` to print it out for signal `sig`.

However, some debugging commands, such as the `next` command, and those dealing with scoping, *will not work*, due to the splitting procedure and the ESTEREL compilation algorithms, that make complete cross-referencing almost impossible.

This debug support uses the ESTEREL simulation debug code, which is a symbol table relating source names and internal names for variables. Thus this flag implies the use of the `-simul` flag in the ESTEREL compilation.

## 5 Using XECL

The XECL program is a graphical user interface to ECL. When invoked with no arguments, the main XECL window appears. The “ECL compiler” is really the ECL part (parsing the input file, splitting the code into ESTEREL and C parts, and writing out the results), plus calls to several other compilers depending on the options (e.g. ESTEREL, a C compiler; VCC), XECL makes it easier to manage the multitude of options for the various compilers.

There are four parts to the main XECL window.

1. **Input specification.** The user chooses the input file type (ECL or ESTEREL), the input file location, as well as the output directory location. Note that, in addition to ECL input, the XECL program also accepts ESTEREL input. This makes it easier for the user to run a complete ESTEREL compilation flow: it automatically runs the ESTEREL compiler plus a C compiler with the appropriate options.
2. **Options.** There are two primary groups of options. Tooltips and keyboard shortcuts are available for all options.
  - (a) Compiler modes
    - ESTEREL, which replaces the `-ESTEREL_SIMUL` option in ECL
    - VCC import, which replaces `-VCC` and related options (e.g., `-WORKSPACE`, `-TYPELIB`, etc.)
    - CLEAN
    - TEST
  - (b) Compiler options
    - CHECK and Debug (`-G` in ECL), as in ECL
    - Pop-up windows with option lists for ECL advanced options, ESTEREL options, and VCC import options.
3. **Command lines.** As options are entered, the command-line flow is shown in a separate box (for example, the calls to the ESTEREL and C compilers in the case of a request for ESTEREL simulation output). In this box, the user can modify all the command lines (for example, to change the C compiler). Note that this capability does not exist in ECL: the options to the compilers that are run are determined by the ECL compiler and cannot be changed by the user. To the right of the command-line box, are several buttons Run (run a compilation), Rebuild (rebuild the command lines based on the checked options, discarding any modifications typed in the command-line box by the user), Clear (clear all options except the input file and output directory), Save (save the current set of checked options to a file), and Load (load a set of options from a file).
4. **Output.** The output of a compilation is shown in this box.

### Caveats

- VCC import is not available when the input is an ESTEREL file. The reason for this is that the ESTEREL file is not parsed, and in order to create the VCC interface files, one needs to parse a file and read its inputs and outputs.

- Cutting and pasting from XECL windows when running on Windows2000 does not seem to work.
- The button for moving up a directory in the file browser does not seem to work on Windows (a bug to the authors of this code has been reported). However, using the pull down menu below the field with the current directory allows the user to select the directory above. Once this is done, the button for moving up a directory works.

## 6 ECL Syntax

See the files in `ecl/test/*.ecl` and `ecl/test/regression/*.ecl` in the distribution for examples. ECL syntax is the same as C, with the addition of reactive constructs. Thus, we concentrate on these new constructs.

In this section, *sig* is a signal, *sig\_exp* is a signal expression, *exp* is an expression (also called a data expression), *stmt* is a compound statement, *i* is an integer variable, *var* is an integer or float variable or the value of a signal with type integer or float.

### 6.1 File Naming

At ECL filename typically has the extension `.ecl`, though this is not necessary. Given an input filename `filename.ecl` (or `filename`), the ECL compiler will create or overwrite the files `filename.h` and `filename.str1`, thus files with these names should not be created by the user.

### 6.2 Modules and their Interfaces

The basic functional block in an ECL design is a module. The syntax for specifying modules is similar to ANSI C, and is as follows:

```
Module :=
  module module_name( interface_declarations )
  {
    stmt;
  }
```

An ECL program is translated to an ESTEREL file with C side functions. Therefore, its behavior is determined by the ESTEREL semantics, which determines by program analysis which module is the main module; the ESTEREL compiler also defines its own main function. As a result, the word `main` as well as the name of any other C standard library function should not be used for `module_name` or defined as a function in an ECL program. If this is done, the compiler chain may not produce a warning and the resulting executable will probably not produce the desired results. No keywords from C or ESTEREL should be used as variable or module names in an ECL file.

Modules are passed arguments through the interface declaration which can be signals, parameters, or variables:

**Signals** : all signals are defined by a kind and a type. Allowed kinds are input, output, sensor. Type can be any C type plus “pure” (meaning that the signal does not carry a value).

**Variables** : these are like variables in C, and are distinguished from signals by having no kind. Since Esterel does not have the notion of variable arguments to a module, variable arguments of the top-level module will appear in its interface as *sensors*, that are assigned to a *variable* immediately inside the module. In this way, it is possible to pass them as parameters to ESTEREL modules, they are updated only when the module is first run (while ESTEREL sensors are updated at every instant), and it is legal to assign new values to them (while ESTEREL sensors cannot be assigned).

For example, the following ECL code:

```

module abs_value (output int s, int i) {
    if (i < 0) i = -i;
    emit (s, i);
}

```

generates the following ESTEREL code:

```

module abs_value: output s : integer; sensor i : integer;
var sens_ecl_i := ?i : integer in
    if sens_ecl_i < 0 then
        sens_ecl_i := -sens_ecl_i
    end if;
    emit s( sens_ecl_i )
end var
end module

```

In VCC top-level variable arguments of a module become block *parameters*, whose value is set by the simulator at initialization time.

The precise syntax for the module arguments is as follows:

```

interface_declarations :=
    input type input_name
|
    output type output_name
|
    sensor type sensor_name
|
    type variable_name

```

```

type :=
    legal-c-type
|
    pure

```

### 6.3 Statements

The body of an ECL module is a composite statement, including C statements and reactive statements. The syntax of the statements is summarized here, with further details following in Section 7 (local signal declarations are allowed only at the beginning of blocks, as C block-level variables):

```

statements :=
    all the C statements;
|
    halt; // Halt statement
|
    pause; // Pause statement
|
    signal [type] sig [= init_value ]; // Local signal declaration
|
    await( [sig_exp] ); // Await statement
|
    emit( sig [, value] ); // Emit statement
|
    present( sig_exp ) // Present statement

```

```

    stmt;
  [else
    stmt;]
|
sustain( sig[, value] );           // Sustain statement
|
do                                 // Abort statement
  stmt;
abort( [sig_exp] )
[handle
  stmt;]
|
do                                 // Weak abort statement
  stmt;
weak_abort( [sig_exp] )
[handle
  stmt;]
|
do                                 // Suspend statement
  stmt;
suspend( [sig_exp] )
|
par stmt;                          // Parallel stmts, must be at least two
par stmt;
|
fork {                              // Same as above, but may have only one
  stmt;
} join;

```

## 6.4 Data Expressions

A data expression *exp* can be used in any context where one would use it in C, such as in assignments, conditional statements, and so on. It may contain a mix of signals and variables, where the appearance of a signal name implies a reference to the signal *value*. The operators in expressions can be divided into 3 categories: unary, binary, and ternary. Each of these can be further divided into 2 categories: those which can be used only in C, and those which can be used in either Esterel or C.

Summary of data operators for variables and signal values:

**Unary operators for C only** : *\*var*, *&var*, *~ exp*, *++i*, *--i*, *i++*, *i--*, (type) *exp*, *sizeof exp*, *sizeof (type\_name)*

**Unary operators for Esterel or C** : *- exp*, *! exp*, *pre(exp)* (or *@(exp)*) (which are written in Esterel respectively as *- exp*, *not exp*, *pre(?exp)*). The **pre** operator, whose shortcut is **@**, returns the status or value in the previous execution. The pre operator must be applied directly to a signal (not to a signal expression, and not to a field of the data type of the signal). The following is legal: *pre(s)[10]* (and all parenthesis are required). The following is illegal: *pre(s[10])*.

**Binary operators for C only** : *<<*, *>>*, *&*, *^*, *|*, *+ =*, *- =*, *\* =*, */ =*, *% =*, *<< =*, *>> =*, *& =*, *^ =*, *| =*, *a[b]*.

**Binary operators for Esterel or C** : *\**, */*, *%*, *+*, *-*, *<*, *>*, *< =*, *> =*, *==*, *!=*, *&&*, *||*, *=*, which are written in Esterel respectively as *\**, */*, *mod*, *+*, *-*, *<*, *>*, *< =*, *> =*, *=*, *<>*, and, or, *:=*.

**Ternary operators for C** : *exp 1 ? exp 2 : exp 3*

## 6.5 Signal Expressions

A signal expression *sig\_exp* may contain only signals, where their appearance implies a test on their presence/absence status. The signal operators are `&`, `|`, `~` which are written as `and`, `or`, `not` in Esterel. Entire signal expressions, tested by reactive statements (i.e., not sub-expressions), may be qualified with the keyword `immediate` or with a `delay` expression. The signal expression syntax is as follows:

```
sig_exp :=
  immediate(simple_sig_exp);
|
  delay(integer_expr) [simple_sig_exp];

simple_sig_exp :=
  1; /* This is the Esterel 'tick' */
|
  0; /* This is 'not tick' */
|
  sig;
|
  ~simple_sig_exp;
|
  simple_sig_exp & simple_sig_exp;
|
  simple_sig_exp | simple_sig_exp;
|
  pre(simple_sig_exp);
```

As discussed in Section 7.6, the `immediate` keyword implies that the expression is tested by the enclosing reactive statement (such as `await` or `abort`) immediately, rather than waiting for the next execution. The `delay` keyword obtains the value, say *n*, of the integer expression, and then returns true when the signal expression has been true for *n* executions.

## 6.6 Always-true Signal Expressions

In some cases where a `sig_exp` is accepted, one can omit the signal expression completely, and the always-true signal expression will be used. (This is equivalent to the `tick` in ESTEREL.) The always-true signal expression can be used as follows:

```
await( );
abort( );
weak_abort( );
delay(delay_expr)
```

In the last case, all delay expressions can be followed by the empty signal expression, e.g. `await(delay(3))`.

## 6.7 Delay Expression

A delay expression is any valid integer expression. However, since a delay expression must be translated entirely in Esterel, no C-specific operators or function calls should be allowed (see also the Esterel manual for specific limitations on the delay expressions).

## 6.8 Module instantiation

A module can be instantiated inside another module by simple calling the module name as in a function call. For example, to instantiate module A twice in module B:

```

/* Define module A */
module A( input int in1, output int out1 ) {
  ... statements ...
}

/* Define module B */
module B( ) {
  /* Define internal signals for the connections */
  signal int internal1;
  signal int internal2;

  /* First instance of module A */
  A( internal1, internal2 );
  /* Second instance of module A */
  A( internal2, internal1 );
}

```

A module can be defined after its use. In that case the compiler will issue a warning when the use of a module is first encountered. To avoid the warning, and in those cases when the instantiated module is found in a different file, a module can be declared and not defined (prototyped). The prototype consists of the usual module declaration, followed by a semicolon. For example, to prototype module A above:

```
module A( input int in1, output int out1 );
```

There are two differences with respect to ANSI C function declarations:

1. A module cannot be declared `extern`, because ESTEREL does not permit modular compilation yet (this restriction may be lifted in future versions).
2. Argument names cannot be omitted and must match those of the definition, i.e. the following are both illegal:

```

module A( input int, output int);
...
module B( input int in1, output int out1);
...
module B( input int in2, output int out2) { ... }

```

## 7 Notes on Semantics

### 7.1 Module Execution

The basic functional unit of an ECL description is the module, and each ECL description has a top-level main module. The operation proceeds as follows:

- The main module is repeatedly called to compute. The calling may be done by an RTOS, or controlled by a global clock.
- Each time the module is called, it performs a computation by
  - reading its current inputs
  - executing beginning at the previous halt point and continuing to the next halt point
  - emitting its outputs

## 7.2 Halting

The **halt** statement stops execution of this module for this computation, and for every computation thereafter. The only way of proceeding from a **halt** statement is if it is enclosed in a preemption statement, and the preemption condition becomes true.

The **pause** statement stops execution of this module for this computation; computation is continued on the next call.

## 7.3 Signal Waiting

The signal waiting statements include **await**(*sig\_exp*) and **await**(*sig*). The first implies that the program halts the current computation and doesn't continue until the next computation in which *sig\_exp* is present. The second is equivalent to **pause**.

## 7.4 Signal Emission

The signal emission statements are **emit**(*sig*) and **emit**(*sig*,*value*). In both cases this implies that the signal is emitted during the current computation. There is no assumed ordering of the emissions if more than one signal is emitted in a computation.

The **sustain** statement causes the continuous emission of a signal. That is, **sustain**(*sig*) implies that *sig* is emitted, and execution halts here and continues in the next computation by again emitting *sig*. This continues until the statement is preempted.

## 7.5 Signal Test

The testing of signal statuses is done with the **present** and **pre** constructs. When control reaches this point in an ECL specification, the presence of the signal for this computation is used to determine the subsequent action (the then or the else clause). The status of the signal in the previous execution can be obtained using the **pre** construct.

## 7.6 Expressions

The immediate qualifier on a signal expression implies that this expression should be evaluated and acted upon immediately, rather than halting and waiting until the next module computation. For example, **await**(*a*); will always wait until the *next* computation in which the signal *a* is present, while **await**(**immediate**(*a*)); will continue with this computation if *a* is present in this computation.

A delay expression that qualifies a signal expression implies that the signal expression must be present a number of times equal to the evaluation of the delay expression.

Given a signal name in an ECL file, how do we know if it is the signal presence/absence status we are referring to or its value? Recall in Esterel, the value is accessed using the '?' notation, that is, *?sig* returns the value of *sig*. In ECL, the '?' is not used since the context of a signal name provides unambiguously information about whether it is the presence status or the value that is required: all reactive statements refer to signal statuses, all others to signal values. Some examples:

- presence/absence: **present**(*sig*), **present**(**pre**(*sig*)), **do** {} **abort**(*sig*) (all preemption statements),
- value: **if** (*sig*), **if** (**pre**(*sig*)), **do** {} **while**(*sig*) (all math operators on signal values).

## 7.7 Preemption

The **abort** preemption statement operates as follows. Each time control reaches the **abort** statement (whether for the first time, or after a halt for some subsequent computation), the

preemption condition is checked, and if true, control continues immediately after the end of the **abort**.

The **weak\_abort** statement is similar except when the preemption condition becomes true, the **weak\_abort** statement executes anyway up until it reaches a halting statement, at which point control continues immediately after the **weak\_abort** statement.

The **suspend** statement is different. Each time it is reached, its preemption condition is checked, and if true, the statement is not executed and control remains precisely where it was at the end of the last computation. It is related to the *history* construct in StateChart-like hierarchical FSM languages.

## 7.8 Concurrency

The ECL syntax supports two totally equivalent forms of specification of concurrency, by means of the **par** statement prefix or by means of the **fork/join** compound statement. Both imply that the compound statements specified to operate concurrently will be analyzed by the Esterel compiler to ensure their unique behavior regardless of the ordering of the statements between parallel branches, and subsequently written in straight-line C code.

## 7.9 Looping

Looping constructs are exactly as in C (**for**, **do**, and **while**), and so were not mentioned in Section 6. Note that the preemption construct, such as **do { ... } abort (sig\_exp);**, despite the use of **do**, *do not* imply a loop. They can, of course, contain a loop, and this leads to a richer set of loops in ECL than is found in standard C.

Consider the following examples:

**C loop** : **do { ...; } while(exp);**. The looping condition must be a C expression (not a signal expression), and there may or may not be waiting constructs in the body. The exit condition is checked only when it is sequentially reached during an execution (i.e., at the bottom of the **do** loop), and the appropriate action is then taken. C loops must be translated to ESTEREL if they contain waiting statements, and must be translated to C if not.

**preemption loop** : **do { while (1) { ...; } } abort(sig\_exp)**. The interior **while** loop must have a waiting statement somewhere in its body. Once control is inside the **while** loop, at each execution the preemption condition (**sig\_exp**) is checked *first*, and the loop code executed only afterwards as appropriate (if the loop was not preempted). Preemption loops must be translated to ESTEREL.

**C loop with signal expression termination** : **do { ...; present(sig\_exp) i = 0; else i = 1 } while(i);**. This **do-while** loop terminates exactly during the execution in which **sig\_exp** is true and when the **present** statement is sequentially reached. The similar preemption loop above terminates in *any* instant in which **sig\_exp** is true, and terminates *immediately* without executing any of the code inside the loop.

## 8 Miscellaneous Topics

### 8.1 Code Splitting

The ECL compiler splits a program into C code and ESTEREL code. It is worth understanding how this splitting is done, so that if a particular behavior is desired, different constructs could be used to direct the compilation in the right direction.

The ECL code is first parsed into an abstract syntax tree. In the tree, each node is either a statement, an operator or a leaf node that represents an immediate value (a constant) or a reference to an identifier. Each node in the tree is labeled with an attribute. There are currently three values for the attribute:

E: this node must be translated into ESTEREL because there is no C counterpart. Examples are the `await` statement, preemption statements, and all the constructs that manipulate signals.

C: this node must be translated into C because there is no ESTEREL counterpart. Examples are the operators used in data type manipulation.

EC: this node can be translated into either ESTEREL or C. In this case there is no requirement to translate one way or the other. Examples are conditional statements, loops (though loops may require `await` statements if translated in ESTEREL; at this time, the ECL compiler doesn't check that the condition is satisfied, but if it is not, the ESTEREL compiler catches it later) and operators common to both languages.

Because ESTEREL code is able to call procedures written in C using a function or procedure call, it is possible to embed C nodes within (in the parse tree terms: below) an E node by translating it into a function, removing it from the parse tree and replacing it with the corresponding function or procedure call. On the flip side, C code is unable to refer to ESTEREL code: hence, everything below a C node (i.e. part of the source contained in the C node) must necessarily be translated in C.

It is clear then that the initial subdivision into E, C and EC nodes may be inconsistent: a C node may have an E node below, or an E node may have a C node above. In these cases, the ECL compiler generates an error message. Otherwise, the compiler continues and assigns all EC nodes to either E or C nodes. The procedure that is followed is the following: during the first step, the C attribute of a node *a* is propagated below to all the nodes in the tree that have *a* as a transitive parent; this ensures that no ESTEREL code is called from the C code. In a second step, the E attribute of a node *b* is propagated above to all its transitive parents; this ensures that all ESTEREL code falls within other ESTEREL code. Finally, all remaining EC nodes are transformed in either E or C nodes, depending on the user's choice (see the command line options).

This flow is a little more complicated by the fact that certain statements are linked to other statements and they have to be translated both in the same language. In particular, breaks and continue statements must have the same translations as the corresponding containing loop.

Once the E and C attributes have been computed, ECL continues by generating corresponding functions for all the C nodes, and replacing these nodes with an Esterel function call. By definition then, the original code is pure ESTEREL code that calls the C functions in the appropriate place. At this point, the code is written to the output files.

## 9 Application Notes

This section contains a collection of application notes which address several issues in further detail.

### 9.1 Common Problems

This section lists a few common error messages, what they mean, and their solutions.

**\*\*\* Error: variable being read before written...** This message comes from the ESTEREL simulator after compiling with `-ESTEREL` and starting the simulator. Try starting the simulator with the command `filename.exe -novarcheck`. See section 3 and an example in section 2.4.2.

**Error: storage class ;signal; for x not allowed...** Signals declarations can only be local or on the interface of modules.

**Syntax error at line ... of "acomp:..."** The ECL compiler first runs the preprocessor, the parses the result. A curly-brace mismatch in the original file, in particular a missing closing brace, could cause this error.

## 9.2 On Causality and Loops

The following program:

```
while (1) {
  for (x; y; z) {
    await(S);
  }
}
```

is disallowed (it will result in a causality error in the ESTEREL compilation). The ECL and ESTEREL compilers cannot determine if the `for` loop is executed at least once, so it could be combinational or sequential, and since it is within another loop, this could cause a zero-delay loop (if the `for` loop is combinational).

The user may be able to re-write this code, either maintaining the intended behavior or slightly modifying it, to create an acceptable program. Here are two solutions.

1. The program can be simply modified to ensure that sure the global loop has at least 1 halting statement. Note that in this case the behavior is modified:

```
while (1) {
  await();
  for (x; y; z) {
    await(S);
  }
}
```

2. The user may know that the loop is executed at least once, and thus there is no zero-delay loop. In this case, the specification should be written so that it is obvious that the loop is executed at least once. The ECL compiler will ensure that the body of the loop has a halting statement. For example, if the original user specification is as follows:

```
while(1) {
  for (i = 0; i <= 10; i++) {
    await(IN);
    emit(OUT, IN);
  }
}
```

it should be re-written as follows:

```
while(1) {
  i = 0;
  do {
    await(IN);
    emit(OUT, IN);
    i++;
  } while (i <= 10);
}
```

## 9.3 On Causality and Data/State

Modeling state in ECL and ESTEREL programs can be tricky, since state is meant to be derived and not modeled directly. Nonetheless, users tend to do this using variables or the value part of signals. Users also share data between (possibly concurrent) portions of programs. This inevitably leads to causality error messages from the Esterel compiler. Esterel is very conservative in its causality checking, especially when it comes to pure data (variables, loops, etc.).

Certainly, one can use the value of either a variable or a signal to model data/state. In Esterel, variables cannot be shared in parallel branches (except read-only), and signals cannot be read (previous cycle value) and written (current cycle value) in one transition: they must have one consistent value for each transition. A causality error will result in either of these cases.

If a piece of data is accessed by two concurrent blocks of code, there are two interesting cases to consider:

- One block reads and writes the data, the other only reads it,
- Both blocks both read and write the data.

In the first case, causality errors in Esterel can be avoided without introducing a time delay by using both a variable and a signal. In the second case, the only way to avoid causality errors is to hide the fact that the data is being written by the two modules by using procedure calls to write the data. While Esterel will no longer complain of causality problems, the resulting design is non-deterministic and its behavior dependent upon the static scheduling of the concurrent blocks as determined by the Esterel compiler. Furthermore, the concept of data or state that is accessed by two concurrent blocks is also less explicit to the designer, and will lead to designs more difficult to understand and debug. Still, one can do this with caution.

Examples of the two cases follow.

**Case 1** Module M has two parallel branches called M1 and M2. M1 reads and writes the value of `first`, while M2 merely reads it. The original ECL specification follows. Note that `first` is a signal so it can be shared between M1 and M2. Since it is both read and written by M1, an `await()` was added by the user to get around the causality errors.

```
module M (input int data_in, input pure Tick, input pure RESET,
         output pure GOT_DATA_IN, output pure START_JOB,
         output int NEW_REQUEST)
{
  signal int first;

  do {
    while (1) { /* main loop */
      emit(first, 1);
      par while (1) { /* M1 begins here */
        await(data_in);
        emit(GOT_DATA_IN);
        if (first) {
          emit(START_JOB);
          await(); /* Added to avoid causality problem */
          emit(first, 0);
        }
      }
      par while (1) { /* M2 begins here */
        await(Tick); /* Some outside system clock */
        if (!first) {
          emit(NEW_REQUEST, 1);
        } else {
          emit(NEW_REQUEST, 0);
        }
      }
    }
  } abort(RESET);
}
```

Note that if the first input received is both signals `Tick` and `data_in`, `NEW_REQUEST` will be emitted with the value 0 because of the `await()`. This is not the correct behavior.

In the second version of this program, `first` is implemented with a variable `first_state`, which is used to read and write the value in M1, and a signal `first`, which is used to transmit its value to M2:

```
module M (input int data_in, input pure Tick, input pure RESET,
          output pure GOT_DATA_IN, output pure START_JOB,
          output int NEW_REQUEST)
{
    signal int first;
    int first_state;

    do {
        while (1) { /* main loop */
            emit(first, 1);
            first_state = 1;
            par while (1) { /* M1 begins here */
                await(data_in);
                emit(GOT_DATA_IN);
                if (first_state) {
                    emit(START_JOB);
                    emit(first, 0);
                    first_state = 0;
                }
            }
            par while (1) { /* M2 begins here */
                await(Tick); /* Some outside system clock */
                if (!first) {
                    emit(NEW_REQUEST, 1);
                } else {
                    emit(NEW_REQUEST, 0);
                }
            }
        }
    } abort(RESET);
}
```

**Case 2** Now suppose M2 also writes the value of `first`. In this case, we use only the variable `first_state`, we allow it to be read in M1 and M2, and we hide the writing act by implementing it with a small macro that is defined in a header file to be included in the final compilation – Esterel only sees it as an external function, and it is in-lined during C compilation.

```
/* aux.h */
#define set_first_state(a, b) (a = b)

/* M.c */
module M (input int data_in, input pure Tick, input pure RESET,
          output pure GOT_DATA_IN, output pure START_JOB,
          output int NEW_REQUEST)
{
    int first_state;

    do {
```

```

while (1) { /* main loop */
    first_state = 1;
    par while (1) { /* M1 begins here */
await(data_in);
emit(GOT_DATA_IN);
if (first_state) {
    emit(START_JOB);
    set_first_state(first_state,0);
}
    }
    par while (1) { /* M2 begins here */
await(Tick); /* Some outside system clock */
if (!first_state) {
    emit(NEW_REQUEST, 1);
    set_first_state(first_state, 1);
} else {
    emit(NEW_REQUEST, 0);
}
    }
}
} abort(RESET);
}

```

In the modified example, M1 toggles `first_state` if it is 1, and M2 toggles `first_state` if it is 0. It is undefined what will happen if `Tick` and `data_in` occur in the same instant. The current Esterel compiler happens to schedule M1 followed by M2, probably just by program ordering.

## 9.4 User-defined Types

As in C, the user can define and use types. In order to use these types with the ESTEREL simulator, special functions must be written so the simulator can convert the type to and from a textual representation, and to check that a legal type has been entered. For more information on how this is done, see the ESTEREL documentation, or the example `ecl/test/regression/audio_buffer.ecl`. The definitions for these special functions can be placed in the ECL source file, and surrounded with `#ifdef STRL_SIMUL ... #endif`, which is defined when the `-ESTEREL` flag is given to the ECL compiler.

## 9.5 Initializing Variables

ESTEREL and C have different semantics for variable initialization. The ESTEREL simulator normally stops whenever it reaches an uninitialized variable and generates an error message. Compiled C code (not from ESTEREL) will run without stopping, using garbage values for any uninitialized variables.

Unfortunately the detailed error checking provided by the ESTEREL compiler and simulator clashes with several aspects of the ECL semantics and compilation process. First of all, initialization checking cannot be performed for non-basic types, because the ESTEREL simulator does not have any visibility of their internals. Secondly, the ECL compilation process may pass to an extracted C procedure a variable before it is initialized. While the C procedure may initialize it inside and thus cause no harm, the ESTEREL simulator assumes that this is an illegal use.

In summary, the `-novarcheck` option of the ESTEREL simulator should be used whenever no variable initialization checking is desired by the simulator (i.e., the C semantics). In some cases, with limited amount of variable initializations performed in extracted pieces of C code, the user can leave the checking on and provide some additional guarantee of code correctness.

For examples of how initialization is handled by the ECL compiler for many different flavors of initialization, see the file `ecl/test/regression/init.ecl`. Run `ecl` on this file and view the results in `init.strl init.h`.

## 9.6 Using Constants

The proper way to use a constant is illustrated by the following ECL program:

```
module test_submodule(const int c, input pure i, output int o) {
    while (1) {
        await(i);
        emit(o, c);
    }
}
module test_const(const int foo, input pure IN, output int OUT) {
    test_submodule(foo, IN, OUT);
}
```

The `c` interface constant in module `test_submodule` and the `foo` interface constant in module `test_const` will become constants in the resulting C code. The value of the `foo` constant should be defined either with a `#define` in a separate header file that is not included in the ECL file but is manually included by the ESTEREL-generated C file, or as a global variable in the ECL file as follows:

```
int foo = 37;

module test_submodule(...) {
    /* same as previous example */
}

module test_const(...) {
    /* same as previous example */
}
```

If it is not defined, the ECL compiler will complete with no errors, but the constant will be undefined at link time. If the constant is defined within this ECL file, the pre-processing step in the ECL compiler will replace it, then generate an ESTEREL file that will not parse correctly.

## 9.7 Empty Modules

Currently, a module declaration and a definition of a module with no behavior both produce a module declaration. For example, the following two lines result in module declarations:

```
module foo();
module bar() {}
```

The way to define a module with no behavior in ECL is to give it a body with an empty statement:

```
module bar() {};
```

## 9.8 Module Arguments

Consider the following declaration:

```
module mymod (input int a, output int b, int c)
```

Clearly `a` and `b` are signals. Our interpretation of `c` is that it should be read once and only once each time `mymod` is called, whether `mymod` is the top-level module or a submodule. The ECL compiler thus translates `c` into an ESTEREL sensor, but also renames it locally so that its value is only read once per invocation. In order to retain this semantics if `mymod` is the top-level module, one must implement the interface of `mymod` and the rest of the system correctly. In VCC, for example, we have implemented `c` as a parameter to the module (all sensor arguments to the top-level module become module parameters).

## 9.9 Debugging Reactivity

If the ESTEREL compilation gives the message "potentially instantaneous loop", you must look at the line and column number of the generated ESTEREL file to find the problem. All loops translated into ESTEREL must have a halting statement (such as `await`) somewhere inside. If such a loop is created with a halting statement, then surrounded by a pre-emption statement, then surrounded by a loop, note that the interior loop may not be executed the first time, if the pre-emption condition is present. Thus, the `await` statement is never executed, and this results in an instantaneous loop. See `ecl/test/regression/causalnot1.ecl` for an example.

If you get a message about reactivity not being allowed in a pure C part, and you are sure you have awaits in all your reactive loops, check the `file.str1` file to see how the translation is done. You may find a called procedure; if so, check the `file.h` to see how that procedure was created. This will make it easier to figure out why your "reactive" loop was interpreted as data. It may be an error in bracketing.

## 9.10 On Esterel Traps

ECL doesn't have a syntax for writing the ESTEREL `trap` statement, which is handy for a block of code to kill itself midstream. The functionality can easily be duplicated using the weak abort command. Suppose the user wishes to write the following code:

```
trap T in
{body}
end
```

where obviously the statement `exit T` appears in the body to kill the block. This can be written as

```
signal T in
  weak abort
{body : substitute "exit T" with "emit T; await tick"}
  when T
end signal
```

Since every `emit T` is followed by a pausing statement, even though the block is given its "last wills" before being aborted, it effectively stops executing right after the `emit T` statement that was encountered.

## 9.11 ESTEREL Keywords

No ESTEREL keywords should be used in naming modules or variables in an ECL file. The ESTEREL keywords are as follows:

**Declaration related:** `module`, `input`, `output`, `inputoutput`, `relation`, `sensor`, `return`, `var`, `signal`, `in`

**Type related:** `type`, `boolean`, `integer`, `float`, `double`, `string`

**Constant related:** `constant`, `true`, `false`

**Signal related:** `combine`, `with`, `await`, `immediate`, `emit`, `sustain`, `present`, `else`, `pre`, `not`, `and`, `or`

**Control flow related:** `loop`, `each`, `every`, `do`, `upto`, `abort`, `weak`, `suspend`, `when`, `trap`, `exit`, `handle`, `nothing`, `pause`, `halt`, `if`, `elsif`, `positive`, `repeat`, `times`, `case`, `do`, `end`

**Function related:** `function`, `call`, `procedure`, `exec`, `task`, `run`

## 10 Acknowledgements

Our thanks to Raphael Boucher who contributed the XECL code; this work was done during his internship at Cadence Berkeley Laboratories in 2001.

## References

- [1] L. Lavagno and E. Sentovich. ECL: A Specification Environment for System-Level Design. In *36<sup>th</sup> DAC*, pages 511–516, June 1999.