Abstract

ECL is a language for control-dominated system-level design that builds upon the Esterel semantics foundation, and adds C-style syntax, data types and control structures. Some extensions are necessary for users to be able to use ECL in large-scale hardware design and synthesis. In this project 8 new constructs are added for RTL hardware modeling to ECL, following guidelines for Esterel hardware extensions developed by Berry and others. These extensions include bit vectors, matrices, equational notation and so on.

1 Introduction

System-level designs are typically conceived as a set of communicating processes. The wide variety of characteristics and requirements of the processes implies that there is no single language that can be efficient for specification. Nonetheless, it is desirable to be able to specify such designs in an integrated environment, so that the design as a whole can be both treated with a common semantics, at least at the communication level, and automatically synthesized, at least to the extent possible.

A framework in which different parts of an embedded system or system-on-a-chip specification, to be implemented on heterogeneous hardware and software resources, can co-exist thinks to this common inter-process communication semantics is described in [3]. It assumes that processes communicate via signals using various buffering and synchronization mechanisms. It also assumes that function, communication and performance of a system are kept as separate as possible.

ECL is a specification language for processes, which are the communicating entities that describe the functionality of the design. It is especially targeted at control-intensive processes. ECL combines two existing languages: a synchronous control-oriented language called Esterel [4] and C, which is already widely used for embedded system programming.

With the growing requirement for hardware feature support in large-scale hardware design and synthesis, the current system design languages need to be enhanced in the ability to handle hardware design. As we have mentioned, ECL, as Esterel, is a control-oriented language and extensions are necessary to enable it to be used in hardware design. In 2000, G. Berry, etc, proposed several hardware extensions to Esterel language that are considered necessary for hardware design [2]. The extensions are a new let constructs that allows the users to embed Lustre-like declarative definitions into Esterel. Specifically, the extensions include:

• the let statement. This new statement “let let-unit-list end” makes it possible to mix the orthogonal styles of data-flow languages such as Lustre, and of imperative languages such as Esterel. Intuitively, the list of equation-style let-units keeps being executed during the lifetime of the statement, which can be preempted by any preemption constructs.
• signal vectors and encoders. Esterel make a very strong distinction between the control path and the data path. The definition of complex types and data operators are left to the host language. However, in hardware design, some data manipulation clearly belong to the control path, such as reading a machine instruction, splitting it into opcode and address fields. Two extensions are proposed to fit this goal. One is signal vectors, with operations such as slicing, reversal, etc. The second is a new kind of control signal, the encoder. It encodes a number using some encoding such as binary, Gray, one-hot, or a user-defined custom encoding.
• non-determinism. Esterel is a deterministic program. But non-determinism is essential to model the environment in which the program evolves, which is essential in circuit verification or optimization. The technique chosen here is a non-deterministic emit statement.
• improved module structure and interfaces. The Classic Esterel module structure is quite rudimentary. Some extensions are proposed to better fit the needs of hardware design.
These extension ideas inspired the hardware enhancement on ECL. In this project, we added some new constructs to ECL to handle hardware design following the guidelines of hardware enhancements in Esterel.

We organize this report as following: Part 2 gives an overview on ECL as a background introduction. Part 3 presents the new constructs for hardware enhancement considered in this project. We show the implementation results of these new constructs in Part 4. Finally we conclude the report and summarize our work.

2 Background

ECL Overview

The basic syntax of an ECL program is C-like, with the addition of the module. A module is like a subroutine, but may take special parameters called signals, which carry both event presence/absence status information and a value. An orthogonal, “kernel” subset of Esterel constructs are provided in ECL to manipulate the signals.

The ECL compilation process has three phases.

1. An ECL file is parsed and split into
   • a control-dominated, reactive part that is mapped to an Esterel source file, and
   • a data-dominated, data-oriented part that is mapped to a C source file, and
   • a “glue logic” part that allows Esterel statements to access fields of ECL non-scalar data types, and which can be mapped to a variety of application-dependent implementation languages.

2. The native Esterel compiler [5] translates the Esterel source to an EFSM.

3. The ESFM is compiled into an optimized software (C) or hardware implementation (VHDL or Verilog) [6].

ECL Statements

The statements are the same as in C. The following statements and signal access functions are added:

1. emit (signal, value);
2. await (signal_expression);
3. halt ( );
4. present (signal_expression) statement1 else statement2;
5. do statement1 abort (signal_expression);
6. do statement weak_abort (signal_expression);
7. do statement suspend (signal_expression);
8. par (statement1; statement2; ...);

They are very similar to the Esterel language.

3 New constructs for hardware extensions

Several new constructs are considered for hardware extension of ECL. They are inspired by the hardware extension ideas of Esterel and can be divided into several groups:

1. For the let-during construct. The let construct has the basic form

   ```
   let
       let-unit-list
   end let
   ```

   The meaning is: let-unit-list is a list of let-units that will execute in parallel until the let statement is preempted by some other Esterel construct. The let-units can be of 3 kinds: functional definitions of signals, permanent imperative statements, and verification observers.

   The let statement can be realized by the parallel statement fork-join in ECL. We introduce a new kind of constructs equations for signal definition.

   ```
   X <=> Y & B;
   Y <= C | D;
   Z <= B & C ? B + C - pre(Z);
   ```

   **Figure 1. Equations**

   The meaning is as follows:

   • The status of Y is implied by that of expression “C | D”. It is emitted whenever “C | D” is true. The signal Y can also be present for other reasons.

   • Similarly, the status of Z is implied by (B & C). The only difference is that Z is a valued signal and its value is given by B + C – pre(Z).

   • The status of X is equal to that of Y and B during the life time of the let statement. Therefore, X will be emitted if Y and B are both present, and must not be emitted by any other part of the program otherwise.

The permanent imperative statement is preceded by the always keyword:

```
always emit T;
always run M;
```

**Figure 2. Always**

T is always emitted. It is equivalent to the imperative sustain instruction. An module in always statement is executed repeatedly at each instant, and is restarted afresh at next cycle.
Another kind of let-unit is observers. They are defined on the fly as arbitrary expressions within the code. They are only emitted when a property violation is detected. No new constructs are needed for observers since they can be realized by the present statement.

2. For signal vectors and encoders. There are two types of signal arrays:

\[(\text{signal int}) \ X[10];\]
\[\text{signal (int } X[10]);\]

where the former is an array of 10 signals each with type integer and the latter is one signal with type array of 10 integers. A new construct called \text{generate} is introduced for defining or emitting array elements, or reversing arrays.

\[
\begin{align*}
\text{signal int } f[10]; \\
\text{fork} \{ \\
\text{generate } i < 9 \\
\text{present (} f[i]\text{)} \text{ emit (} f[i+1], \text{ } i); \\
\text{endgenerate} \\
\} \text{ join;}
\end{align*}
\]

\textbf{Figure 3. Generate}

Generate is actually a static evaluated loop. To avoid writing an explicit for loops in trivial cases, we provide a mechanism for slicing arrays. A \text{slice} of a signal array is given by a continuous subrange of indices. Slices of arrays are used in the \text{array definition} which defines an array or slice to be equal to an array expression.

\[
\begin{align*}
(\text{signal int}) \ A[10], B[2]; \\
A[8:9] = B;
\end{align*}
\]

\textbf{Figure 4. Array definition}

Encoders are another important extension. An encoder is a signal that carries an encoded number. Three standard encodings should be provided: binary, gray and one-hot. The basic ideas are to provide encoding and decoding functions as part of a library but further discuss is necessary to obtain a clearer definition of this part of extension.

3. For improving module structures. In hardware ECL, arrays can be used as parameters of modules and we introduce \text{template modules} to permit implicit declaration of parameters.

\[
\begin{align*}
\text{template <type T, int i> module m} \\
\text{(input T f[i]) (.....)} \\
\text{module q(...) \{} \\
\text{signal char g[10];} \\
\text{m<char, sizeof(g)/sizeof(char)>} \\
\text{(g);} \\
\text{\}}
\end{align*}
\]

\textbf{Figure 5. template modules}

Such modules can be handled since we will know array sizes and types at compile-time and these implicit parameters become explicit.

Another improvement of module structures is to enable argument initialization, as in C, C++, and Java.

\[
\begin{align*}
\text{module foo (input int a = 2,} \\
\text{output struct \{ int b, c\) d = \{1, 3\}} \\
\text{\}}
\end{align*}
\]

\textbf{Figure 6. Argument Initialization}

4. For interfaces. In hardware ECL, we may have really big interfaces. It is no longer practical to declare the interface of a module within the module. We make interfaces first-class objects and introduce hierarchy of interfaces. We use keyword \text{port} for the declaration of interfaces.

\[
\begin{align*}
\text{port Intf \{} \\
\text{input int X;} \\
\text{output Y;} \\
\text{input value int foo;} \\
\text{\};} \\
\text{port Intf2 \{} \\
\text{input int Z;} \\
\text{port Intf intf;} \\
\text{port Intf intf2;} \\
\text{\};} \\
\text{module m (port Intf p; port Intf} \\
\text{^p1, \ldots)} \{ \}
\text{port Intf2 t;} \\
\text{\ldots} \\
\text{\}
\end{align*}
\]

\textbf{Figure 7. Ports}

The basic interface declaration is identical to the Classical ECL module interface declaration. The mirror of an interface \text{p}, denoted by \text{^p}, is the interface obtained by changing the polarity of the input and output signals.

We can see that hierarchy is used in interfaces. The field of an interface is permitted to be another interface. Interfaces can be hierarchically nested to arbitrary depth.

Interfaces are also useful in importing externally defined circuits. Such circuits would be considered as modules without bodies and their interfaces can be defined as above.

5. Register types. The next statement implements the register behavior in the enhanced hardware ECL. The following two statements are defined to have the same functionality of assigning the next value of variable \text{b} to be 12 when signal \text{a} presents.

\[
\begin{align*}
present a \{ \text{next (b, 12)};\n\end{align*}
\]
next b <= a ? 3;

**Figure 8.** next statements

A fork structure is created in the esterel output with the definition of each register used in the module parallel with the other translation of the statements in the module.

## 4 Implementation

In Part 3 we introduced 8 new constructs for hardware enhancement in ECL. They have been successfully added to the ECL compiler. We present the techniques for implementation in this part.

There are two possible ways to implement a new ECL construct. One is to use the combinations of existing ECL statements to express it. This method is preferred since it is cheap. Another way is to create new special file for parsing the new construct. This should only happen when the functionality of new construct cannot be expressed by available statements. Fortunately, all these new constructs can be reduced to existing structures in ECL.

### 4.1 always

The syntax for Construct always is as:

```plaintext
always { statement_list; }
```

An always statement is reduced to:

```plaintext
For (; ;) {
  statement_list;
  await();
}
```

**Figure 9.** always implementation

### 4.2 equations

The syntax for equations is as:

- `signal_identifier <=> logic_signal_expression;
- `signal_identifier <== logic_signal_expression;
- `signal_identifier <== logic_signal_expression ? expression;

We have shown an example in Figure 2 so we do not go to details about the syntax of `logic_signal_expression` and expression. Equations are also reduced to existing statements by the parser.

We give the parsing result of Figure 2 in Figure 11.

```plaintext
present (Y & B) emit (X);
else present(X) {
  fprintf(stderr,"Assertion X
  <=> Y & B violated\n")
  exit(-1);
}
for (;;) { present (C | D) emit
  (Y); await(); }
```

**Figure 10.** Implementation of equations

Note that an error will be printed when violation is found.

### 4.3 signal arrays and generate

We mentioned that there are two types of signal arrays. Currently only the first one is implemented. The way to compile signal arrays is to unroll them totally into separate signals since they are not accepted by Classical Esterel. For example,

```plaintext
(signal int) f[10];
```

is compiled to:

```plaintext
signal int f[0];
...
signal int f[9];
```

The reference to an array element is compiled to the corresponding new signal. (This leads to the result that the array index is only permitted to be constant.) Because of the same reason, generate construct cannot be reduced to a for loop (also they have the same functionality) but has to be totally expanded. Such expansion is difficult to be implemented directly in the compiler. We added a pre-processing step to expand the generate statement and thus the compiler only has to unroll the signal array.

Currently, the syntax for generate is as:

```plaintext
#generate identifier constant_integer
  statement_list;
#endgenerate
```

The compiling result of the example in Figure 3 is:

```plaintext
present (f_0) emit (f_1);
present (f_1) emit (f_2);
...
present (f_8) emit (f_9);
```

**Figure 11.** compiling result of generate

Nested generate statements cannot be handled currently due to the technique of using pre-processing step. Further improvement is needed for this statement.

### 4.4 array definition

The syntax of array definition is as:

```plaintext
array1[constant1: constant2] = array2;
```

The two arrays can be either standard arrays or signal arrays. For standard arrays, a for loop will be generated by the parser. An example is shown in Figure 13.

```plaintext
a[8:9] = b;
```

is replaced by:

```plaintext
{ int _t; for (_t = 0; _t < 2; _t++) a[_t+9] = b[_t]; }
```

**Figure 12.** Implementation of array definition
For signal arrays, this for loop has to be expanded totally.

4.5 template modules
Template modules are also handled by the pre-processing step since it would be hard to implement them directly in the compiler. An example of template modules is shown in Figure 14.

```plaintext
#template module m Ti
  (input T f, output char o[i]) {
    ..
  }
#endtemplate

#instantiate m (m12, char, 10)
module t (input char g, .. ) {
  signal char o1[10];
  ..
  m12 (g, o1);
  ..
}
```

**Figure 13.** An example of template modules
After pre-processing an instantiation of the template module will be created:

```plaintext
module m12(input char f, output char o[10]) {
  ..
}
```
which is a normal module declaration.

4.6 argument initialization
This construct allows the arguments to be initialized in the module parameter declaration.

```plaintext
module foo (input int a = 2, output struct { int b,c; } d = { 1, 3} ) {
  int v;
  v = a + d.b + d.c;
  ..
}
```
is translated into

```plaintext
type _type_123;
_type_123 function _init_func_123();
module foo:
  input a := 2 :integer;
  output d := _init_func_123() : _type_123;
  var v:integer in
    v := a +
    DOT_integer_S_0_no_name_b(?d) +
    DOT_integer_S_0_no_name_b(?d);
  ..
```

**Figure 14.** An example of argument initialization

4.7 ports
Ports are realized in ECL as a special kind of structures. Each field of a port must be an input, output or a signal. Since Esterel does not support such kind of structures, we have to unroll each port into separate signals. An example is shown in Figure 16.

```plaintext
module m (port Intf p, port Intf ^p1) {
  port Intf2 t;
  emit (p1.x, 3);
  emit(t.intf.x, p.x + 1);
  emit(t.intf.foo, p.foo);
}
```
is compiled to:

```plaintext
module m (input int p_x, output p_y, input value int p_foo , output int p1_x, input p1_y, output value int p1_foo) {
  signal int t_z,
  signal int t_intf_x,
  signal int t_intf_y,
  signal value int t_intf_foo,
  signal int t_intf2_x,
  signal int t_intf2_y,
  signal value int t_intf2_foo;
  emit (p1_x, 3);
  emit(t_intf_x, p_x + 1);
  emit(t_intf_foo, p.foo);
}
```

**Figure 15.** Implementation of ports
We use the definition of ports in Figure 7. In a word, we have to replace ports by separate signals, one for each port field.

4.8 registered types
The next statement information is stored in the parser tree throughout the parsing of the whole module. A new field called TypeNext with the methods of getTypeNext and setTypeNext are created in the stmtNode class. Each time when one next statement is parsed, the TypeNext field of the compound statement of the module is set to be true. After the whole module is parsed the definition of each of the registers are set to be the parallel statement with the module body.

An example of this construct is the translation from module bar (input pure a, output int b) {
  present a {next (b, 12)};
  await();
  next b <= a ? 3;
in the hardware ECL to

do{   /* abort to keep the original control flow */
  fork{
    /* module body */
    present a {emit (next_b, 12)};
    await();
    present a {emit (next_b, 3)};

    /* module completion */
    emit (_end_of_bar);
  }
  ||
  /* "next" latches */
  {
    always b <= pre(next_b) ? next_b;
  }
} join
} weak_abort(pre(_end_of_bar));
}

in the original ECL/esterel.

Figure 16. An example of register types

4.9 Remaining problems and future work
Some implementation strategies are not satisfiable
yet. For example, signal arrays are just simply
unrolled into separate signals. Such operation causes
some information that can be used in hardware
synthesis to be lost. There seems no easy way to fix
that. Further discussion is needed.

A lot of work is still needed to be done on this topic:
1. Give a clear definition on encoders. Encoders are very important in hardware design and
they must be implemented in ECL as part of hardware enhancement.
2. Improve the implementation of signal
arrays.
3. Handle nested generate loops. In some test
files, nested generate loops are used so they should
be handled by the compiler.
4. Implement the other type of signal
arrays.

5 Conclusion

In this project 8 new constructs for hardware
enhancement are implemented successfully for ECL.

Also a bug in the original ECL language is found and
reported. The future work will be focused on
extending the functionality of the features, improving
the error tolerant ability, and implementing more new
features. We really appreciate this experience of
learning the language parser / compiler construction
and our Java programming skill is improved in this
project. Finally we want to sincerely acknowledge to
the generous help from our mentors Luciano
Lavagno, Ellen Senovich and Roberto Passerrone.

Reference

[1] Luciano Lavagno, Ellen Sentovich. ECL: A
Sangiovanni-Vincentelli. Design of embedded
1997.
esterel. 1996.
H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C.
Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. Hardware Synchronous Programming of