ECL: A SPECIFICATION ENVIRONMENT FOR SYSTEM-LEVEL DESIGN

Gérard Berry
Ed Harcourt
Luciano Lavagno
Ellen Sentovich

Abstract We propose a new specification environment for system-level design called ECL. It combines the Esterel and C languages to provide a more versatile means for specifying heterogeneous designs. It can be viewed as the addition to C of explicit constructs from Esterel for *concurrency* and *pre-emption*, and thus makes these operations easier to specify and more apparent. An ECL specification is compiled into a *reactive* part (an extended finite state machine representing most of the ECL program), and a pure data looping part. The first can be robustly estimated and synthesized to hardware or software, while the second is implemented in software as specified. ECL is a good candidate for specification of new behavior in system-level design tools such as Cadence's Cierto VCC tool[1]. ECL is especially targeted for specification of control protocols between data-computing behavioral blocks.

1. OBJECTIVES

System-level designs are typically conceived as a set of communicating processes. The processes may communicate synchronously or asynchronously, may be control- or data-dominated, may have hard real-time constraints, and may be used in embedded systems. Such a wide variety of characteristics and requirements 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.

J. Mermet (ed.), Electronic Chips & Systems Design Languages, 205–212.

For this reason, we propose the use of a new executable specification environment called ECL. The main idea is to combine two existing languages to create a specification medium that can benefit from the features of both languages and their existing well-developed compilers. In particular, we add the convenient and concise constructs from Esterel for concurrency and pre-emption to C.

A prototype ECL compiler has been completed and is currently being tested and further developed on some industrial examples.

2. BACKGROUND

2.1 ESTEREL

Esterel [5, 4] is a language and compiler with synchronous semantics. This means that an Esterel program has a global clock, and each module in the program reacts at each "tick" of the global clock. All modules react simultaneously and instantaneously, computing and emitting their outputs in "zero time", and then are quiescent until the next clock tick. This is classical finite state machine (FSM) behavior, but with a description that is distributed and implicit, making it very efficient. This underlying FSM behavior implies that the well-developed set of algorithms pertaining to FSMs can be applied to Esterel programs. Thus, one can perform property verification, implementation verification, and a battery of logic optimization algorithms.

The Esterel language provides special constructs that make the specification of complex control structures very natural. It is often referred to as a *reactive* language, since it is intended for control-dominated systems where continuous reaction to the environment is required. Communication is done by broadcasting signals, and a number of constructs are provided for manipulating these signals and supporting concurrency and signal pre-emption (e.g., parallel, abortion and suspension).

The Esterel compiler resolves the internal communication between modules, and creates a C program implementing the underlying FSM behavior. A sophisticated graphical source-level debugger is provided with the Esterel environment. While Esterel only provides a few simple data types, one can create and use any legal C data types; however, this is separate from the Esterel program, and must be defined separately by the designer. Pure C procedures and functions can be defined by the user and called from an Esterel program, but again there are definitions and code that must be written by hand by the designer.

2.2 C

The C language is ubiquitous. It is used as an application language (system-level programming), used for controlling hardware (e.g., drivers), and also used commonly as a hardware modeling language (e.g., instruction set simulators). However, it lacks control constructs that manage communication and concurrency between modules: one would have to implement these through hand-crafted data types and parameter passing. Nonetheless, C is a widely used language: the user-base is huge, there is much legacy code, and there are many robust compilers.

3. ECL: ESTEREL + C LANGUAGES

3.1 OVERVIEW

ECL is primarily for *authoring new modules* in a system-level design tool. It is expected to be particularly useful for specification of of control-oriented, software-dominated glue communication functions such as protocol stacks. It supports a *mix of control (reactive) and data statements*, and automatically synthesizes the code needed for the interaction of these two. It compiles a *maximum subset* of the ECL specification into reactive modules in Esterel and subsequently into asynchronously communicating extended Finite State Machines called *Codesign Finite State Machines* (CFSMs [3]). CFSMs have a semantics that admits *robust optimization* and *synthesis to either hardware or software*, and their cost and performance can be estimated for a variety of possible subsequent implementations [3]. The rest of the program is compiled to data modules implemented in C and called by the Esterel modules.

3.2 SYNTAX

In terms of coding constructs and style, ECL simply combines Esterel and C. In particular, the concurrent and pre-emptive constructs of Esterel are added to C, along with communication by signals for controlling the flow.

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**. The signals behave as signals in Esterel, and an equivalent subset of Esterel constructs are provided in ECL to manipulate them. As a simple example, the following code fragment:

```
#define data_size 80
typedef struct { int a, b } my_type;
module read_signal_data(input my_type IN_DATA, output int DONE)
{
```

```
    int i, sum;
    while (1) {
        for (sum = i = 0; i <= data_size; i++) {
            await(IN_DATA);
            sum += IN_DATA.a * IN_DATA.b;
        }
        emit (DONE, sum);
    }
}
```

waits for `data_size` occurrences of the input signal, sums the two fields of each occurrence, and emits the `DONE` signal when all have been received. Note that in C, there is no natural construct for the communication through signals as done here; in Esterel, there is no automatic handling of the user-defined data type, and there is no explicit for loop. Though this can be easily specified with Esterel loops, for most designers it is more natural to use C looping constructs. In addition, Esterel loops must be reactive, that is, contain a halting statement. This could lead to an inefficient software implementation of loops that purely walk through, say, an array without waiting for external inputs, because it would require an FSM transition for each iteration that is more suited for hardware, rather than a straightforward loop-based software implementation.

3.3 SEMANTICS

The semantics of an ECL program are synchronous for each standalone, top-level single reactive module, just as with Esterel or any extended finite state machine language. At a higher-level, the modules are interconnected and will communicate via the semantics imposed at this higher level. In our current CFSM-based back-end implementation, a globally asynchronous semantics is applied at this network level.

The communication between parts of an ECL program, whether it be synchronous (within a top-level module) or asynchronous (between modules), is always done through signals (which may be valued). The decision about how to partition the design into synchronous individual modules communicating asynchronously is an implementation issue. We currently leave it to the designer to make such a choice, based on simulation and exploration at the specification level to aid in choosing the best implementation.

3.4 SUPPORT FOR PURE C AND ESTEREL

An ECL program typically is a mix of C and Esterel-like statements, but pure ANSI C and pure Esterel (with C-like syntax) are supported as subsets of ECL. This implies that legacy C code can be used in ECL-based system design.

The current compilation scheme for ECL translates as much of an ECL program as possible into Esterel, for full synthesis and optimization. In this way, we also maximize the subset of ECL that can be implemented as hardware, by being translated completely to Esterel first and CFSMs later. It is a subject for future work to explore schemes (more oriented towards legacy code handling and software implementation) in which only a minimal part of ECL, including only some reactive constructs (such as **abort**) is translated in Esterel, and the rest is left as C.

3.5 ECL COMPILATION

The prototype ECL compiler being implemented has two primary parts:

1. **Parser/Reactive Recognizer:** parses the ECL input into an internal data structure; traverses this data structure to recognize the reactive parts (Esterel-based statements), separate them, and write the result out in the form of C and Esterel code. This basically generates a description that can be used by the Esterel compiler to generate a top-level reactive FSM calling some (residual) C code. As previously stated, a maximum subset of the ECL program is compiled into Esterel, since it is this portion that can be estimated, aggressively optimized, and synthesized to hardware or software.

2. **Esterel compilation:** This part has two possibilities. The first is using the Esterel compiler to generate C code. Here, the code generated can be generic C code, or can be targeted for simulation using one of the Esterel simulators, or can be targeted for import to a commercial system design simulation tool. This flow has been tested with different simulators on different platforms and for a variety of test examples. The second possibility is using the Esterel compiler to generate CFSMs. This is a new part of the Esterel compiler being developed by the Esterel team in France and Cadence to compile Esterel programs to CFSMs and then to estimatable software and hardware. At a higher-level, the modules are connected via a globally asynchronous network communication scheme. As a side benefit, this compilation path uses the common

DC format, which implies that with it, there will be a connection to other synchronous languages besides Esterel. This second path is currently under development and test.

3.6 ON CONTROL AND DATA

One of the primary tasks of the ECL compiler is to recognize and separate the control and data parts. An ECL program will contain a mix of statements, manipulating signals and ordinary variables, communicating through signals, looping through computations, pre-empting operations, and calling external (C) functions. There are two types of loops: *reactive* loops which contain at least one Esterel await-type statement (e.g. **await (S);**), and *data* loops containing no such statements, and hence appearing to be instantaneous from a signal communication standpoint. A data loop is just like an ordinary loop in C, and is forbidden in Esterel since, with the notion of time, such a loop would be instantaneous (executing the same statements twice in the same clock tick). Data loops are allowed in ECL, but are compiled into separate C (inlined) functions called by the Esterel code.

The only ECL constructs that *cannot* be compiled into CFSMs are thus the *external (user-defined) functions*, and the *data loops*. In the treatment of complex data types, the ECL compiler generates the appropriate type definitions in Esterel and C, as well as the field and element access (inlined) functions called by Esterel and implemented in C.

The example above would be compiled into the following Esterel code:

```
type my_type;
function get_a (my_type): integer;
function get_b (my_type): integer;
procedure AUTOINCR (integer)();
module read_signal_data:
input IN_DATA:my_type;
output DONE:integer;
var sum, i: integer in
   loop
      sum := 0;
      i := 0;
      trap done in
         loop
            if i > 80 then exit done;
            await IN_DATA;
            sum := sum + get_a(IN_DATA) * get_b(IN_DATA);
            AUTOINCR(i)();
```

```
        end loop
      end trap;
      emit DONE (sum);
   end loop
end var
end module
```

A simple C header file would contain a set of macros implementing **AUTOINCR**, **get_a** and so on. In this case, there is no data loop, and hence the pure C code part is empty.

3.7 OPTIMIZATION AND SYNTHESIS TO HARDWARE AND SOFTWARE

A top-level single CFSM is synchronous and equivalent to an extended finite state machine. This implies that all the robust techniques for optimization (combinational and sequential logic optimization, optimization based on reachable states, etc), verification (both property verification and specification/design verification), and synthesis of FSMs can be applied. Furthermore, CFSMs can be synthesized indifferently to hardware or software. Thus, for the *reactive* parts of the ECL program, powerful techniques for implementation generation are available.

4. CONCLUSIONS AND FURTHER DIRECTIONS

Since this paper was originally published, at FDL in 1998, considerable progress has been made. At that time, an ECL compiler prototype, proprietary to Cadence, was under test within their system-level design tools.

At the 1999 Design Automation Conference, a more extensive paper on ECL was published [6]. As of this latest writing, a new version of the compiler has been written in Java and is freely available on the web [2]. In addition, it has a smooth flow for integrating ECL models into the Cierto VCC system-level design tool by Cadence [1].

To summarize the capabilities, the ECL compiler compiles ECL programs into a maximally synthesizable subset; one important current direction for research is to synthesize only a *minimal* subset of the ECL program (the minimal reactive part), while leaving the rest in its C-code specification form. This style of compilation will be useful for importing legacy code, where the user would like to preserve the existing code as much as possible, while adding just enough "reactivity" to break this code into smaller pieces that interact through signals.

References

[1] For more information on Cadence's Cierto VCC product, visit /http://www.cadence.com/technology/hwsw/ciertovcc.

[2] The Java version of the ECL compiler has recently become available. Visit http://www.cadence.com/programs/na/research.shtml and follow the ECL project link.

[3] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publishers, 1997.

[4] G. Berry. *The Foundations of Esterel.* 1998. To appear.

[5] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] E. Sentovich and L. Luciano. ECL: A Specification Environment for System-Level Design. In 36^{nd} *DAC*, pages 511–516, June 1999.