ECL: A Specification Environment for System-Level Design

Luciano Lavagno Ellen Sentovich

Cadence Berkeley Laboratories, 2001 Addison Street, 3^{rd} floor

Berkeley, CA 94704-1103, USA

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 *waiting*, *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, thus nicely supporting a mix of control and data. The reactive part can be robustly estimated and synthesized to hardware or software, while the data looping part is implemented in software as specified.

1 Introduction

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 with a mixed hw/sw implementation. 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.

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 thanks to this common inter-process communication semantics is described in [6]. It assumes that processes communicate via *signals* using various buffering and synchronization mechanisms, which can be efficiently implemented in practically relevant special cases (e.g., when synchronization is static as in Static Data Flow networks [9], or when buffering is bounded and small as in Codesign Finite State Machine networks [1]). It also assumes that *function*, *communication* ("untimed synchronization") and *performance* of a system are kept as separate as possible, by enabling one to

- integrate parts of a system specified using different Models Of Computation, each best suited for an application domain or flavor,
- keep the functional specification and the communication structure independent of the implementation architecture,
- derive, at least in part, timing, power and area figures from a mapping of the functional specification and communication onto the architectural and communication resources chosen by the designer, in order to perform quick architectural exploration.

In this paper we assume that an infrastructure for the specification, analysis (performance, safety, liveness, etc.) and synthesis of embedded systems satisfying the above requirements is available. Several non-commercial prototypes exist, as well as a few industrial developments [5]. This view is also at least partially consistent with those of the System Level Design Language definition committee [11].

In this paper we focus of the definition of a specification language for *processes*, which are the communicating entities that describe the *functionality* of the design. This language is especially targeted at *control-intensive* processes, in which decision and quick reaction to unpredictable inputs dominate over lengthy computations on regular data streams. The language can also be used to specify data-dominated computation fragments embedded within the control structure, and it has a rigorous semantics for this coordination.

The language is called ECL, for Esterel/C Language. 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. We have selected some convenient and concise constructs from a synchronous control-oriented language called Esterel [7]: namely those for waiting, concurrency, and pre-emption. We have added these constructs, with their precise synchronous semantics, to C, which is already widely known and often used for embedded system programming. The resulting language combines the full power of ANSI C and its facility for constructing and manipulating complex data types, with a clean communication model based on the exchange of signals between and within modules. Since Esterel has an *extended Finite State Machine (EFSM) semantics*, which lends itself to both hardware and software implementation, ECL can also be used to evaluate different hardware/software partitioning trade-offs [1].

The choice of C as the main language is, of course, somewhat arbitrary. C++ and Java could have been reasonable alternatives. C is just a pragmatic choice for the time being, for the following reasons:

- C *today* is better suited for embedded system design. Moving from assembly language to C is already a big step for a large community of designers.
- The approach can easily be extended, whenever the need arises, to any other language.

The key contribution of this paper, however, is the idea of adding *truly synchronous reactivity* to an existing, widely used language.

Related Work A number of recent works have attempted to leverage the broad knowledge of C in the embedded system design arena, and its apparent suitability to implement complex embedded software. They all started from the observation that C in itself does not satisfy all the requirements of a clean design methodology in this field, since it lacks constructs to specify the *interaction of concurrent modules*, and the reaction to high-priority events and exceptions. Most of these works have used the *reactive language* family as a source of inspiration for how to specify all these notions.

The Reactive-C language [4] takes an approach that is very close to ours, in that it extends C with reactive Esterel-like constructs. However, its implementation scheme relies on direct compilation to C, thus yielding a less clear semantics (some RC statements have a non-intuitive meaning) and an inefficient, interpreted implementation.

The Scenic language [10] also inherits some constructs (waiting for signals and aborting computations in the presence of signals) from Esterel, but it implements them as C++ classes, thus also reducing the efficiency of the implementation. Moreover, Scenic does not take the full step towards a truly synchronous semantics (in which computation must behave as if it took zero time with respect to the environment), which is more deterministic and intuitive than its approximations.

The SpecCharts language uses StateCharts [8] as the control specification mechanism, and extends it with the ability to specify computations in C within each state and on transitions between states. This requires the designer to work in a mixed graphic/textual environment, and suffers from the problem that the StateCharts semantics is not exactly synchronous and not always precise.

Our Approach Our approach relies on proven compilation technology from CMA (Sophia Antipolis, France) in order to provide truly synchronous deterministic semantics, coupled with state-of-the-art software and hardware synthesis techniques.

There are two main differences between our approach and that of existing thread-based concurrent programming paradigms, like Java.

1. ECL modules use signals, rather than procedure/method calls or shared variables to communicate. Signal detection and emission, with its inherent ability to model reaction to multiple sources explicitly and succinctly, seems
   - better suited than procedure calls to model complex concurrent specifications such as those that arise in communication refinement.
   - easier and safer to use than shared variables (even protected by semaphores, monitors, and so on) to specify communication and synchronization among cooperating tasks.

   The ECL signal is conceptually closer to the event flag or mailbox synchronization services offered by several RTOSs, but is much more integrated with the language structure itself than those services.

2. The control structure of a top-level ECL module is collapsed as much as possible into a single EFSM, thus maximizing the performance of the synthesized software. In this sense, the ECL compilation process, and the choices between collapsing as an EFSM and extracting as a C procedure, can also be viewed as performing a trade-off between control and data-oriented implementation.

The latter is also a profound difference from *asynchronous* concurrent languages such as OCCAM and Lotos, in which atomicity is guaranteed only at a very low level (individual synchronization primitives and elementary statements), instead of the potentially larger amount of atomic computation performed in an ECL/Esterel *instant*. Larger atomic units (in fact, with user-selectable size) help the understandability and *verifiability* of a specification, possibly at the expense of code size and execution time.

Another difference from OCCAM, Lotos and ADA is the nature of the communication primitive. The rendezvous used by OCCAM, Lotos and ADA is complex and expensive to implement in a heterogeneous distributed architecture. Signal-based communication, on the other hand, is reasonable to implement both synchronously and asynchronously, in hardware, in software and at the boundary.

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*. The signals behave as signals in Esterel or VHDL: they carry both "event" presence/absence status information and a value (signals carrying only one of the two are also allowed in Esterel and ECL). An orthogonal, "kernel" subset of Esterel constructs (detailed in Section 4) are provided in ECL to manipulate the signals.

The ECL compilation process has three phases.

1. An ECL file is parsed and split (according to heuristics that will be detailed in Section 4) 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 (e.g., C or VHDL).
2. The native Esterel compiler [2] translates the Esterel source to an EFSM.
3. The EFSM is compiled into an optimized software (C) or hardware implementation (VHDL or Verilog) [1].

If the data-dominated C part is empty, then the complete ECL specification can be implemented either in hardware or in software. Otherwise, only software is *currently* an implementation option (hardware implementation becomes also an option, of course, by using high-level synthesis).

The remainder of the paper is organized as follows. In section 2 we review the basic features of the Esterel and C languages. In section 3, we describe the ECL environment. In section 4, we illustrate the ECL syntax and efficient specification capabilities with an example. Finally, our current work on applying ECL in industrial design flows is described in section 5, and conclusions in section 6.

2 Background

Esterel Esterel [3] 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 to write, understand and compile into EFSMs (and hence either software or hardware). 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 waiting, concurrency and signal pre-emption (e.g., await(signal), 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 definitions and code must be written by hand by the designer. ECL automates this task, by automatically generating all the required declarations and definitions ("glue code").

3 ECL Environment

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 carry a presence/absence status (and also may carry a value). The decision about how to partition the design into synchronous individual modules communicating asynchronously is an implementation issue. We currently leave it to designer to make such a choice, based on simulation and exploration at the specification level to aid in choosing the best implementation.

ECL Operational Semantics The synchronous semantics of individual ECL modules implies that there is a difference between the execution model of C and that of ECL. In C it is a sequence of *statement executions*. In Esterel, and hence ECL, it is a sequence of *instants*. In each instant, the top-level ECL module (like the `main` in C) receives a snapshot view of its input signals, with presence/absence status and value information. At this point, that is execution of a function or module, C and ECL differ. In ECL, the control part (mostly statements from Esterel) then computes *instantaneously* which internal and output signals are present for the current instant. That is, even though the computation may be specified in several steps, Esterel compiles away these steps to a single FSM transition that is assumed to take *zero time*[1]. Then the data part (including calls to extracted pieces of C code) is executed depending on this signal presence/absence information.

In the next instant, execution of the module begins *where it left off at the end of the previous instant*, for example when it reached an `await` statement. (All statements where an instant ends and the next one begins, like `await` and `halt` are called *halting statements* in Esterel.) Thus the state of a module is stored implicitly in its halting statements, whereas it would have to be stored explicitly with variables in a procedure implementation (e.g., in C) of the same concurrent behavior.

Compilation The ECL compiler front-end uses a standard C/C++ parser to parse the ECL input into an internal data structure. It then traverses this data structure to extract the reactive parts (Esterel-based statements) and write the result out in the form of C code, C header and Esterel files. The header and Esterel files are used by the Esterel compiler to generate a top-level reactive FSM written in C (that in turn calls the C code generated by the ECL front-end).

Since ECL is a mix of C and Esterel-like statements, one can envision using the ECL environment to specify designs in which the modules may be written in ECL, C only, or Esterel only. Furthermore, since many constructs in Esterel are themselves very C-like, one has a choice in the compilation phase of ECL when splitting behavior into the reactive part and the data part. Furthermore, a subset of pure ANSI C[2] (C-only) and Esterel-only (with C-like syntax) specifications are supported as subsets of ECL. This implies that legacy C code can be used in ECL-based system design. Caution must be used in this latter case, of course, because the compilation from ECL to an EFSM has the potential benefit of making a reaction to events much faster than in hand-written code (due to the capability of the Esterel compiler to do case analysis much better than a human designer for large specifications). However, this speed-up comes at a price, that is the potential explosive growth of code size. The designer can exert manual control over portions of the code mapped to Esterel and C, and hence alleviate this problem.

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 EFSMs 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.

Key Features We complete the summary of ECL by highlighting its main features:

- ECL is a combination of C and Esterel-like reactive statements, giving the designer a familiar language with a few new constructs to ease the specification of control.
- ECL nicely handles mixed control/data specifications, with a control portion that has fully synchronous semantics, and a data portion that has the familiar C semantics.
- The control portion is equivalent to an EFSM, permitting the use of existing powerful techniques for optimization, analysis, and synthesis of FSMs. In particular, logic synthesis and optimization can be applied to reduce size or improve speed, implicit state exploration techniques can be used for optimization and functional analysis, and synthesis techniques used to create implementations in hardware or software [3, 1].
- ECL compilation involves a choice when splitting the code to the reactive part (fully synthesizable) and the data part (software-only, and possibly preserving the form of the incoming code). An ECL prototype compiler is currently implemented and under test on industrial examples.

ECL Statements The statements are the same as in C, to maximize reuse of existing code. We have only added the following statements and signal access functions for manipulation of signals and their values. Esterel programmers will immediately notice the similarities to that language.

1. `emit_v(signal,value)` defines that `signal` is present in the current instant and simultaneously defines its value. The statement `emit(signal)` is used for pure signals.
2. `await (signal_expression)` ends the current instant and waits for the occurrence of the given expression in some later instant. A `signal_expression` involves only signal names and Boolean operators (`&`, `|`, `~`).

[1] This is in sharp contrast, for example, with the VHDL execution model in delta time, in which signal updates performed at the current delta cycle are seen only at the next delta cycle, and hence delta computation takes *unit delay*.

[2] Currently there is no way to support global and static variables, due to the strong Pascal-like scoping rules of Esterel. Work to lift this limitation is under way.

If one needs to split a loop into multiple instants, without actually waiting for any signal, one can use the `await()` statement (with an empty expression). This statement introduces a sort of "delta cycle" in the ECL module execution, i.e. it causes the execution to continue in the next instant, but keeps the module active regardless of its input events (normally a module that has reached the end of an instant "sleeps" until one of its input signals has an event)[3]. This mechanism can also be used to force a loop to be implemented as a sequence of EFSM transitions, instead of being extracted as C code.

3. `halt()` stops the execution of the module, until preempted. (It generally makes sense only inside an abort statement, which can be interrupted from outside.) An instant for a module ends when it (and all its instantiated submodules) reaches a `halt` or `await` statement.

4. `present(signal_expression) statement1 else statement2` performs `statement1` if `signal_expression` returns true (a non-zero value) and `statement2` otherwise.

   Note that a signal name appearing in a `signal_expression` (i.e., being tested in a reactive statement) implies a test on the presence of the signal. In any other context it refers to the value of the signal. Thus, the statement

   ```
   present (A) {
       if (A) then emit(OUT);
   }
   ```

   will emit `OUT` if `A` is *both* present and has a non-zero value. In a way, signal names are *overloaded* in ECL: they imply presence in the context of `signal_expressions` tested by reactive statements, and value in the context of normal C-style expressions used in assignments.

5. `do statement1 abort(signal_expression)` executes as follows. When control reaches it, it starts executing `statement1`, until it completes execution or reaches a `halt` or `await` statement. If `signal_expression` becomes true in any later instant when `statement1` has not yet completed execution, then `statement1` is not allowed to perform any action for that instant, and control is passed immediately to the next statement after abort.

   Suppose that one wants to define a piece of code that must be executed only when the statement terminates due to the occurrence of `signal_expression`, as opposed to normal termination, when control just reaches the end of `statement1` (this is like the `catch` clause in Java). The `abort` statement allows an optional handle clause as follows:
   `do statement1 abort (signal_expression)`
   `handle statement2`

6. `do statement weak_abort (signal_expression)` is similar to `abort`, but it allows `statement` to execute for the instant in which `signal_expression` is true, and terminates it only at the end of the current instant.

7. `do statement suspend (signal_expression)` is similar to abortion, but only temporarily stops `statement` from executing in any instant in which `signal_expression` is true. In analogy to UNIX, abort is like `^C` while suspend is like `^Z`.

8. `par { statement1; statement2; ... }` concurrently executes the statements (often sub-module executions) included in the `par` statement.

   Shared signals between parallel statements are admitted, as long as only one statement is doing the writing. This is the same semantics that is used by the Esterel compiler, and is safe thanks to synchronicity. Every reader sees the value of the signals in the previous instant, while updates by the writer are performed only at the end of an instant (like in synchronous edge-triggered circuits). Shared variables, as in Esterel, may be either only read by the parallel statements, or have a single writer and no reader.

9. Module instantiation is syntactically equivalent to C procedure call.

4 An example: ECL at work

In this section, we illustrate the ECL syntax and flow with an example. The example contains a header information with constants and user-defined data types, three computation modules, and one top-level module running the submodules concurrently. We illustrate along the way the ease with which one can conceive and specify inter-module communication by thinking in terms of the communicating objects (signals), rather than abstract states and transitions. Consider first the type declarations and the module declaration in Figure 1.

The module has two input signals: `reset` is pure and thus carries only event presence/absence status information, and `in_byte` carries both a status and a value of type `byte`. The only output signal is a structured type. Note the use of the C union construct to model two possible views of the packet, for different layers in the protocol stack.

The module has two local variables, `cnt` and `buffer`. Initially control passes inside the loop and the `abort` statement. The module then halts waiting for the first `in_byte`. It assembles the `PKTSIZE` bytes, and transmits them to the next stage by means of the `outpkt` signal. The module is restarted whenever the `reset` signal is present, because control is passed from the `await` statement (the only halt point inside the `abort` in this case) directly to the end of the outermost for loop.

Let us consider now two other modules which are used in our protocol stack fragment, and pictured in Figures 2 and 3.

There are two types of loops in ECL, as shown in the previous examples.

1. *Reactive* loops which contain at least one halting statement (e.g. `await (in_byte)`) in each path (see, for example, the `for` loop in Figure 1). Such loops are compiled to Esterel loops.

2. *Data* loops containing no such statements, and hence appearing to be *instantaneous* from a signal communication standpoint (see, for example, the `for` loop in Figure 2). Data loops are allowed in ECL, but are compiled into separate C (inlined) functions called by the Esterel code.

The module in Figure 3 shows the combined use of the `par` and `abort` statements. Note how concurrency (`par`) is used to terminate the long, multi-instant packet-related computation (not shown here) only if an error is detected in its header: the long computation is run in parallel with reactive code that catches the `crc_ok` signal, and may terminate the computation with the `kill_check`

[3] The correct implementation of this feature of the ECL language requires the correct interaction with the environment in which the module is placed. If modules are triggered to react based on events only, then a trigger signal must be generated; otherwise, a feature forcing the rescheduling of the module must be used.

```
#define HDRSIZE 6
#define DATASIZE 56
#define CRCSIZE 2
#define PKTSIZE HDRSIZE+DATASIZE+CRCSIZE

typedef unsigned char byte;

typedef struct {
    byte packet[PKTSIZE];
} packet_view_1_t;

typedef struct {
    byte header[HDRSIZE];
    byte data[DATASIZE];
    byte crc[CRCSIZE];
} packet_view_2_t;

typedef union {
    packet_view_1_t raw;
    packet_view_2_t cooked;
} packet_t;

module assemble (input pure reset,
    input byte in_byte, output packet_t outpkt)
{
    int cnt;
    packet_t buffer;

    /* outermost reactive loop */
    while(1) {
        do {
            /* get PKTSIZE bytes */
            for (cnt = 0; cnt < PKTSIZE; cnt++) {
                await (in_byte);
                buffer.raw.packet[cnt] = in_byte;
            }
            /* assemble them and emit the output */
            emit_v (outpkt, buffer);
        } abort(reset);
    }
}
```

Figure 1: An ECL module assembling bytes into packets.

```
module checkcrc (input pure reset,
    input packet_t inpkt, output bool crc_ok)
{
    int i;
    unsigned int crc;

    while (1) {
        do {
            await (inpkt);
            for (i = 0, crc = 0; i < PKTSIZE; i++) {
                crc =
                   (crc ^ inpkt.raw.packet[i]) << 1;
            }
            emit_v (crc_ok,
                crc == (int) inpkt.cooked.crc);
        } abort (reset);
    }
}
```

Figure 2: An ECL module checking a Cyclic Redundancy Code.

```
module prochdr (input pure reset, input bool crc_ok,
    input packet_t inpkt, output pure addr_match)
{
    signal pure kill_check; /* local signal */
    bool match_ok;

    while (1) {
        do {
            await (inpkt);
            par {
                do {
                    /* some lengthy computation,
                       determining the value of
                       match_ok */
                } abort (kill_check);
                {
                    await (crc_ok);
                    if (~crc_ok) emit (kill_check);
                    /* else just wait for both
                       to complete */
                }
            }
            /* now both branches have terminated */
            if (crc_ok && match_ok) emit (addr_match);
        } abort (reset);
    }
}
```

Figure 3: An ECL module performing a computation on the packet header.

```
module toplevel (input pure reset,
    input byte in_byte, output pure addr_match)
{
    signal packet_t packet;
    signal bool crc_ok;

    par {
        assemble (reset, in_byte, packet);
        checkcrc (reset, packet, crc_ok);
        prochdr (reset, crc_ok, packet, addr_match);
    }
}
```

Figure 4: The ECL top-level module for our simple protocol stack.

signal. The CRC check in Figure 2 is run in parallel with this module by the top-level module, which is shown below. This abortion of a long computation is only possible because the computation is surrounded by an `abort` statement and contains code inside that halts; thus the status of the abort signal can be checked periodically. This would not be possible with a normal C block.

The `addr_match` signal may be emitted only after both branches have completed, either normally or through the `kill_check` abortion signal.

The top-level module in Figure 4 executes all three modules in parallel and connects them with two internal signals, `packet` `crc_ok`. Note that its only role is to instantiate concurrent modules. Hence it could be implemented

- *synchronously*, by compiling it using ECL, thus resulting in a single EFSM for the whole protocol stack, or
- *asynchronously*, by simply interconnecting the three ECL modules as processes communicating via signals.

The former choice will yield a more efficient time-performant implementation at the expense of larger code size. The latter choice

Example	Part.	Memory size				Execution time	
		Task(s)		RTOS		Tasks	RTOS
		code	data	code	data		
Stack	1 task	1008	160	5584	1504	4,283	8,032
	3 tasks	1632	352	5872	1744	4,161	8,815
Buffer	1 task	7072	80	7120	3040	51	123
	3 tasks	2544	144	7376	3536	57	145

Table 1: Results of synchronous/asynchronous implementation trade-offs.

can be partitioned between hardware and software (e.g., the CRC computation may be good candidate for hardware), and is more lightweight and less performant. Of course, the behavior of the two may be different in general, e.g., when a `reset` signal occurs and is received at the same time by all modules in the synchronous case, and at different times in the asynchronous case, or when `crc_ok` is false and the long computation must be aborted. The designer is, of course, responsible for ensuring that all the resulting variants of behavior are equally good with respect to the overall system specification.

We consider a significant feature of ECL this ability to mix, with little manual intervention, asynchronicity and synchronicity, and to trade off performance and cost.

As a simple example of this kind of trade-off, we compiled the protocol stack example in Figure 4 and a simple audio buffer controller from a voice mail pager design. In both cases we tried two partitions into tasks,

- as a single Esterel source file, and hence as a single task (*synchronous* implementation), and
- as three source files, implemented as separate tasks under control of a simple real-time kernel (*asynchronous* implementation) [1].

The code and data memory sizes and execution time for a MIPS R3000 processor, in bytes and thousands of clock cycles (using a testbench with 500 packets) are shown in table 1.

In the first example, asynchronous composition resulted in a larger and slightly slower implementation, mostly due to the large RTOS overhead with such a small task granularity. In general, synchronous implementations tend to be larger and faster than asynchronous ones, as shown by the second example.

5 Industrial Applications

A prototype ECL compiler is currently under test on industrial designs. The compiler has been implemented as a research project at Cadence Berkeley Labs, and in conjunction with the Felix Co-design initiative from the Alta group of Cadence Design Systems [5]. ECL has been implemented in this Co-design environment.

ECL testing and experimentation is being carried out both with Felix initiative industrial partners, and with members of the COSY European research consortium. COSY's main mission is to explore methodologies and tools for system-level design of the future. In these experiments, ECL is being used in two ways:

- to specify new behavioral blocks in the system, namely controller modules that coordinate activity between data-processing and other control blocks, and
- to facilitate the migration of existing monolithic code to partitioned code for exploration of implementation tradeoffs; in this case, the ECL communication style is used to re-implement large legacy code blocks as smaller blocks that communicate by emitting and awaiting interface signals.

6 Conclusions and Further Directions

We have presented a new language for embedded system specification. It combines the widely known software language C with constructs for waiting, concurrency and preemption from Esterel. It nicely supports specification of mixed control/data modules. The compilation is performed by splitting the source code into reactive Esterel code (as large as possible, in the current implementation) and data-dominated C code. The large reactive portion can be robustly optimized and synthesized to either hardware or software, while the C residual code must be implemented in software as is.

One important current direction for research is to map to Esterel 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. A prototype ECL compiler has been implemented and is being tested on industrial examples.

References

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

[2] G. Berry. *The Constructive Semantics of Pure Esterel*. 1996. To Appear, available now at ftp: //www.inria.fr/meije/esterel/papers/constructiveness.ps.gz.

[3] G. Berry. *The Foundations of Esterel*. 1998. See http://www.inria.fr/meije/Esterel.

[4] F. Boussinot, G. Doumenc, and J.-B. Stefani. Reactive objects. *Annales des Telecommunications*, 51(9-10):459–473, September 1996.

[5] P. Clarke. Felix tools pushed in research project. *Electronic Engineering Times*, October 1998. See http://www.eetimes.com/news/98/1029news/felix.html.

[6] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[7] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[8] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[9] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[10] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the Design Automation Conference*, pages 70–75, June 1997.

[11] System Level Design Language Home page, 1998. See http://www.inmet.com/SLDL/.