

Embedded System Design Specification: Merging Reactive Control and Data Computation

Marco Antoniotti¹ †

Alberto Ferrari †

Luciano Lavagno §¶

Alberto Sangiovanni-Vincentelli †‡

Ellen Sentovich §

(†) PARADES E.E.I.G. Via San Pantaleo 66, I-00186 Rome – ITALY

(§) Cadence Berkeley Labs 2001 Addison St, 3rd Floor, Berkeley, CA, 94704-1103 – U.S.A.

(‡) E.E.C.S. Department University of California at Berkeley, Berkeley, CA – U.S.A.

(¶) Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica

Università di Udine, Udine, UD – ITALY

1 Introduction

Our main goal is to provide a specification mechanism, a language, that allows a system designer to use well-known programming constructs and at the same time to leverage the mathematical properties of the design. We believe the most important criteria to follow for the development of a system level design language are as follows:

- a well-defined semantics for design and verification,
- expressiveness – enough to incorporate both data and control flow information, and
- be close to a well-known programming language that is easy to use.

Two language extensions have been developed that adopt a clear semantic model based on the *synchronous* language ESTEREL in order to handle the issues of reactivity and concurrency. JESTER [1], which extends JAVA, and ECL [7], which extends C. Both languages add reactive constructs with the synchronous semantics of ESTEREL to a base language that is in wide use and has advanced development tools.

The reasons for the selection of C and JAVA as the building blocks for our language are as follows:

- C is ubiquitous in the computing world (it is often viewed as a sort of high-level assembly language), especially in embedded and *Real Time* operating systems (RTOS) implementations.
- JAVA (cf.[8]) has several features that make it interesting for embedded system programming. In addition to being a well-known and easy to use language, JAVA has *threads*, and *synchronization* primitives that can be used to deal with concurrency.

The goal of both languages is to ease the task of specifying and designing the control of embedded systems. The concurrency of the system is expressed by com-

posing the constituent components (classes in JESTER or modules in ECL) either synchronously, according to the semantics of ESTEREL, or asynchronously, according to the semantics of the environment in which the modules are used. The designer is not required to understand the concurrency model supplied by JAVA (for JESTER) or by the RTOS primitives in the C (for ECL), since the semantics are well-defined by ESTEREL and only *implemented* using these primitives. JESTER and ECL have been conceived to be used in conjunction with tools like POLIS and VCC¹. By using both C and JAVA as the basis for our specification methods, we intend to demonstrate that the essential issue is about the mathematical models not the syntax of the language. The syntax of the language matters for ease of use, the availability of development tools and standard policies.

In this paper, the main features of JESTER and ECL are described. Two examples are given to show their applicability.

Related Work. There are many other projects aiming at the extension of JAVA and C with reactive semantics (e.g. [4, 8, 12]). Reactive extensions to C are usually less radical in design; they seek an embedding of “reactive primitives” in (possibly legacy) C code.

Working with JAVA allows greater freedom, inasmuch as the amount of embedded legacy code is limited. The reactive extensions to JAVA all build a layer of classes on top of the regular JAVA environment. The approaches can be grouped in two broad categories: either they construct a sort of “simulation” reactive engine on top of JAVA, or they construct a new scheduling

¹Author current address is: NYU Courant Bioinformatics Group, 719 Broadway 12th Floor, New York, NY, 10003 U.S.A.

¹POLIS [2] and VCC [11] are two tools that provide the user the means to do HW/SW *platform exploration*, i.e., they give the user the ability to evaluate different HW/SW designs of an embedded system. VCC is a commercially available tool and has been developed from the foundations laid out by the academic- and research-oriented POLIS.

machinery based on the `InThread` interface, which provides some reactive primitives for synchronization.

The SystemC proposal recently put forth [10] addresses a similar need, but it uses a C++ library-based approach. Hence the semantics of the language is dictated by what can be achieved within the sequential imperative C++ MOC, and is constrained by the need to refer to the semantics of the Hardware Description Languages that SystemC was designed to abstract.

Another related project is Giotto [5], which is a time-triggered programming language with a compiler and runtime library. It is targeted for control specification of safety-critical applications with hard real-time constraints. The semantics of the Giotto language is only partially synchronous: the *drivers* (which dictate when tasks are called) are synchronous, while the *tasks* take time. The drivers cannot call each other; with this restriction, compilation of these synchronous modules is much simpler than with the full synchrony provided by ESTEREL and thus ECL and JESTER. Thus, Giotto identifies uses a class of synchronous programs that allows efficient schedule synthesis and code generation.

As already mentioned, JESTER and ECL make provisions to generate intermediate formats usable by POLIS and VCC, by means of a separate compilation process targeted for these tools. Therefore, JESTER and ECL can reuse all the machinery supporting the GALS MOC in these tools. While this is possible in principle with the other JAVA and C extensions we saw, to the best of our knowledge, JESTER and ECL are the first systems to make this integration one of their design goals.

2 ECL and Jester

First a brief review of ESTEREL before we give the explanation of ECL, JESTER, and their semantics. The ESTEREL language and formal semantics guarantees that a correct specification is always equivalent to a *deterministic finite state machine*. The ESTEREL compiler provides a means to check a specification for correctness, and to produce a circuit- or automata- or C-code form of the underlying state machine. The machine is actually an *extended* state machine (EFSM), which means there are data actions on the transitions. These data actions are either simple and expressed directly in ESTEREL, or are specified by functions and procedures written by the user in C in another file, and called from the ESTEREL code. Thus, ESTEREL has a link to C-code, but this code is quite separate. The guarantees stated above, equivalence to a deterministic state machine, only apply to the control portion. (For example, the user could write an infinite `for loop` as one of the data actions; clearly this would result in an infinite transition on the machine.) Fur-

thermore, the analysis that the ESTEREL compiler performs must treat data conservatively. Nonetheless, the model provides a powerful method for the specification and compilation of control-dominated machines.

As stated in the introduction, ECL and JESTER are simply the languages C and JAVA with ESTEREL constructs added. These constructs include those for specifying signal communication, pre-emption, and concurrency, which are needed to maintain the ESTEREL synchronous semantics.

The data operations in ECL and JESTER can be done directly in the source file, not in a side file in another language as is done with ESTEREL. In particular, ECL and JAVA allow complex data structures and data manipulations to be specified, and their compilers are knowledgeable about *reactive* (or control) loops and *data* loops. A reactive loop is one which contains at least one halting statement (e.g., `await`), and hence can be analyzed easily by the ESTEREL compiler). These are translated to ESTEREL code by the ECL and JESTER compilers. A data loop is one which does not halt, and thus appears to be instantaneous from a signal communication standpoint. These must be translated to C or JAVA by the ECL and JESTER compilers.

The design that one writes in ECL, for example, looks very much like C code with some extra constructs inspired by ESTEREL. This is translated to an ESTEREL part (which is the “control”) of the module, and a C part, which are the data actions. JESTER is similar. Thus the semantics of the two languages is precisely that of ESTEREL: a design has an equivalent synchronous finite state machine where the control part is guaranteed to be correct, and the data part should be checked by the user or other tools. A simplified view of the compilation process is shown in Table 1. First the ECL/JESTER code is translated into ESTEREL (control) code and C/JAVA code. The ESTEREL code may then be compiled to C (recent developments of ESTEREL allow for the compilation to JAVA as well) for a full software implementation or analysis by simulation, or it can be compiled to VHDL for a hardware implementation. Table 2 shows a summary of the constructs and features of the three languages.

In JESTER and ECL, as in ESTEREL, if a set of modules is compiled together, their communication is synchronous and they are effectively translated to a single EFSM. If the modules are compiled separately, they can be used in a system-level design environment, and their communication will be dictated by the semantics of that environment. This is the GALS semantics for VCC and Polis. JESTER provides this flexibility a little more explicitly: a directive is used in the language to indicate if the module composition is to be synchronous or asyn-

SOURCE FILE	TRANSLATOR	RESULT
ECL source (.ecl file)	→ ECL <i>translator</i>	→ ESTEREL code → C code
JESTER source (.jst file)	→ JESTER <i>translator</i>	→ ESTEREL code → JAVA code → C and JAVA JNI code

Table 1: A simplified view of the JESTER and ECL translation process. The results are passed to the appropriate compilers and linkers to produce executable code.

	ESTEREL	JESTER	ECL
<i>module constructs</i>	module construct	JAVA class	module construct
<i>signal constructs</i>	emit(S(V)) await(S) present test	emit(S, V) await(S) present test	emit(S, V) await(S) present test
<i>control constructs</i>	parallel operator loop sustain S if	parallel block; loop and specially recog- nized JAVA loops sustain(S) if	par block; C loops sustain(S) if
<i>pre-emptive constructs</i>	abort block; when[condition] weak abort block; when[condition] suspend block; when[condition]	abort when(condition) weak abort block; when(condition) suspend block; when(condition)	block; do block; abort(condition) block; do block; weak_abort(condition) do block; suspend(condition)
<i>complex data structures (arrays, objects, structures)</i>	separate low level C interface	as in the full JAVA language	as in the full C language
<i>asynchronous composition</i>	–	compose construct	special environment commands

Table 2: A brief summary of the most salient features of ESTEREL, JESTER and ECL.

chronous. Of course, the behavior of the two types of compositions may be different in general, e.g., when a common signal occurs and is received at the same time by all modules in the synchronous case, and at different times in the asynchronous case. The designer is responsible for ensuring that all the resulting variants of behavior are equally good with respect to the overall system specification. See [6] about techniques for conservatively checking when this equivalence between synchronicity and asynchronicity is satisfied, and [3] for a more powerful (but more restrictive) approach to desynchronization. The ability to mix, with little manual intervention, asynchronicity and synchronicity, according to the way the modules are compiled together, and to thus trade off performance and cost are significant features of of ECL and JESTER.

3 A Jester Example

We used JESTER to implement a specification of a car centralized door lock system obtained from Magneti-Marelli S.p.A, Italy [9]. The original specification was given in natural language plus some timing diagrams.

Its aim is to handle key and infrared inputs to activate the locking and unlocking of a car door.(cf. Figure 1).

The specification requires that each *lock* or *unlock* command issued by the user should be executed only if certain conditions are met. Figure 2 shows a snippet of the `InitialDecoder` module (a simple one for lack of space). This module is in charge of translating the user inputs to an internal representation for the door lock system. A number of reactive loops are made to wait in parallel for user inputs. Once one of the loops detects an input it issues an “internal” command represented by either a `issueOpen` or `issueClose` signal. The `CommandRepetition` module guarantees that the frequency of the commands is not be too high (implementing a “debouncing” module). The module only allows the ten most recent commands in the last 30 seconds to be processed by the remaining part of the system. Figure 3 shows a part of the JESTER class. The queue, `SWQueue`, is normal JAVA code and keeps a maximum of ten commands. The `BatteryCheck` checks that each command can be executed, which is guarantee if the battery voltage is above a 9 V threshold.

The `FinalController` module translates the com-

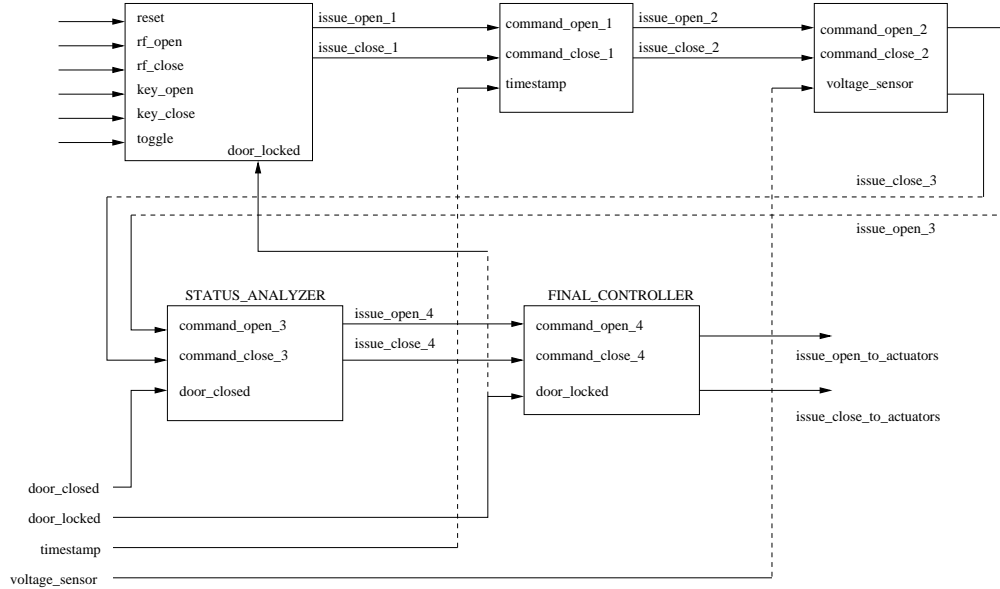


Figure 1: The block diagram for the door lock example. Each block is a separate JESTER class (or module) and the system is composed synchronously according to ESTEREL semantics

```
import jester.runtime.*;

public class InitialDecoder implements Reactive {
    public input signal void rfOpen, rfClose;
    // ... other input and outputs go here.
    public output signal void issueOpen, issueClose;

    public void reaction() {
        parallel {
            loop {
                await(rfOpen || rfClose);
                present(rfOpen) { emit(issueOpen); }
                present(rfClose) { emit(issueClose); }
            }
            // Other loops on other input signals go here.
        }
    } // end of reaction method
} // end of InitialDecoder class
```

Figure 2: A snippet of the InitialDecoder module.

mand to a value for the real actuators, while the StatusAnalyzer based on the status of the door inserts a delay before issuing the command.

We “simulated” the JESTER code by embedding it into a simple JAVA driver whose main aim was to link in a data collection and experiment management JAVA library we developed. The data collected can be manipulated by a regular spreadsheet or stored into a database for future reference. The simulation results agree with the specification and make the model amenable for implementation in either HW or SW.

```
import jester.runtime.*;

public class CommandRepetition implements Reactive {
    {
        public input signal void commandOpen, commandClose;
        public input signal int timestamp;
        public output signal void issueOpen, ...;

        // ... here other Java instances of classes
        public SWQueue myqueue; // a Java queue object

        public CommandRepetition() {
            myqueue = new SWQueue();
        }

        public void reaction() {
            loop {
                await(commandOpen || commandClose);
                present(commandOpen) {
                    boolean slide_window_ok =
                        myqueue.check(timestamp.getValue());
                    if (!slide_window_ok) emit(issueOpen);
                    // ... here some more computation
                } // end of present

                // ... here the part related
                // to 'commandClose'
            } // end of loop
        } // end of reaction() method
    } // end of class
```

Figure 3: A snippet of the CommandRepetition module.

```

        output pure addr_match) {
    signal packet_t packet;
    signal boolean crc_ok;

    par assemble (reset, in_byte, packet);
    par checkcrc (reset, packet, crc_ok);
    par prochr   (reset, crc_ok, packet, addr_match);
}

```

Figure 5: The ECL top-level module for our simple protocol stack.

4 An ECL Example

In this section, we illustrate the ECL syntax and flow with an example. The example contains header information with constants and user-defined data types, three computation modules, and one top-level module running the submodules concurrently. One sees immediately in these examples how C-like the language is, with just a few new key words. We illustrate along the way the ease with which one can conceive of 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 4.

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.

The two other computational modules in this design, `checkcrc` which reads the input and checks that it has arrived correctly, and `prochr`, which processes the header information in an input packet, are not shown here for lack of space. The top-level module in Figure 5 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* or *asynchronously*.

As already mentioned in Section 2, 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.

As a simple example of exploring this kind of trade-off, we compiled the protocol stack example in Figure 5 and a simple audio buffer controller from a voice mail pager design. In both cases we tried two partitions into tasks: (1) as a single ESTEREL source file, and hence as a single task (*synchronous* implementation), and (2) as three source files, implemented as separate tasks under control of a simple real-time kernel (*asynchronous* implementation) [2].

The code and data memory sizes and execution time for a MIPS R3000 processor, in bytes and clock cycles (using a test bench with 500 packets) are shown in table 3. 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 Conclusions

We have presented two language extensions for C and JAVA for embedded system specification, simulation and implementation. The two languages JESTER and ECL build upon the ESTEREL synchronous semantic foundation that provides support for waiting, concurrency and preemption. They nicely support 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 or JAVA code. The large reactive portion can be robustly optimized and synthesized to either hardware or software, while the C residual code must be either implemented in software as is or the user must provide a hardware implementation.

JESTER and ECL serve as a bridge between the “high-level” world of control engineers and theorists and the “low-level” world of embedded systems, by providing a clear semantic model (GALS) that can be leveraged to implement complex control laws in a clearer way. Both JESTER and ECL have been tested on industry supplied examples and are being developed in close contact with our industry partners.

References

- [1] M. Antoniotti, A. Ferrari A. Flesca, and A. L. Sangiovanni-Vincentelli. Jester: An Esterel-based Reactive Java Extension for Reactive Embedded System Co-Design. In *System on Chip Methodologies and design Languages*.

```

#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 4: An ECL module assembling bytes into packets.

Example	Part.	Memory size				Execution time	
		Task(s)		RTOS		Tasks(s)	RTOS
		code	data	code	data		
Stack	1 task	1008	160	5584	1504	4,283,358	8,031,935
	3 tasks	1632	352	5872	1744	4,161,727	8,815,085
Buffer	1 task	7072	80	7120	3040	50,868	123,463
	3 tasks	2544	144	7376	3536	57,117	145,267

Table 3: Results of synchronous/asynchronous implementation trade-offs.

Kluwer, 2001. presented at the Forum on Design Languages Conference (FDL2000), Tübingen, Germany, 2000.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jureska, 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.

[3] A. Benveniste, B. Caillaud, J.-P. Talpin, and Paul Le Guernic. Distributed code generation, desynchronisation, architecture generation, for dataflow synchronous languages: summary of results. Web site www.irisa.fr/sigma2/benveniste/pub/B_a199.html.

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

[5] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. *Giotto: A Time-Triggered Language for Embedded Programming*. Lecture Notes in Computer Science. Springer-Verlag, 2001.

[6] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli. Efficient Methods for Embedded System Design Space Exploration. In *Proceedings of DAC '00*, 2000.

[7] L. Lavagno and E. Sentovich. ECL: A Specification Environment for System-Level Design. In *36th Design and Automation Conference*, pages 511–516, June 1999.

[8] C. Passerone, R. Passerone, C. Sansoé, J. Martin, A. Sangiovanni-Vincentelli, and R. McGeer. Modeling Reactive Systems in Java. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, pages 15–19, Seattle, WA, U.S.A., March 1998.

[9] Magneti Marelli S.p.A. Door lock system specification. Proprietary internal document, 1998.

[10] Web site www.systemc.org.

[11] Information about the Cadence Design Systems VCC tool can be found at www.cadence.com/eda_solutions.

[12] J. S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Richard Newton. Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement. In *Design Automation Conference*, 1998.