Synchronous Programming of a Real-Time Microcontroller

Charles ANDRE Marie-Agnès PERALDI

LASSY, équipe de l'I3S - Université de Nice-Sophia Antipolis / CNRS
41 Bd Napoléon III, F 06041 NICE Cedex, FRANCE

Abstract

The synchronous approach is a new and promising way to design Real-Time systems. We have applied the synchronous programming style to the design of microcontrollers. The programming style is ESTEREL, an imperative synchronous language and the target processor is RTX2000 a high-performance real-time microcontroller.

In this paper, we describe the compilation from ESTEREL down to FORTH(RTX programming language), and we analyze the efficiency of the implementation.

1: Introduction

The design of smart controllers is a critical issue for an ever-increasing number of industrial applications.

These systems are **real-time reactive systems:** They are event driven and "they have to respond to externally generated input stimuli within a finite and specified period"[1]. For this kind of applications, designers must take account of constraints. The most important ones are *safety* and *temporal constraints*. A safety constraint guarantees that nothing harmful may happend. A temporal constraint ensures that reactions are made on time, neither too early, nor too late. Systems with imperative deadlines are called **hard real-time systems**. Another characteristic is the inherent *concurrency* of these systems which results from the interaction between parallel agents.

Controllers are more and more complex to design. Errors may have disastrous consequences. So, for programmable controllers, such as microcontrollers, a powerful and safe programming style is mandatory. Powerful languages can capture sophisticated constraints. Safe programming reduces the gap between "what you program" and "what is executed".

The traditional approach considers all the agents as asynchronous tasks which communicate interactively. Often, a Real-Time Operating System like (OS9, XEC86[2], ...) is in charge of the task management. This solution is *difficult to maintain and to debug* because of the non-deterministic behaviour of a preemptive scheduling.

A second approach gives greater place to asynchronous languages including tasking. However, they are *heavy to use*, their real-time primitives are generally *too weak*, and their performances are not *fully predictable*.

In view of these problems, researchers have provided a new programming style, based on a synchronous approach using so-called "synchronous languages". A synchronous language, like ESTEREL, is endowed with real-time primitives and supports communication and synchronization mechanisms. It is especially well suited for reactive system programming. It is noteworthy that a synchronous language can reconcile concurrency with determinism, leading to predictable behaviour. Last but not least, it can be compiled into efficient code (finite automaton). And yet, many designers have no confidence in synchronous languages because there are based on the perfect synchronism hypothesis. This hypothesis, that implies instantaneous computation, is deemed to be unrealistic.

Our opinion is that a synchronous language, with its simplifying hypothesis, makes the programming of real-time application easier and clearer. To know whether or not the synchronism hypothesis is valid, is typically an implementation issue and application dependant.

The microcontroller we use is RTX2000. It is a high-performance chip, based on a RISC architecture. Several of its characteristics are valuable for real-time applications. However, its programming language (FORTH) may disconcert some users.

Our approach tries to take advantage of both synchronous languages and modern microcontrollers. We advocate the following design method:

1. start with an imperative synchronous language (ESTEREL) to program the application,

2. compile ESTEREL into the machine language of an efficient microcontroller (RTX2000),

3. and then, verify if the implementation meets the perfect synchronism hypothesis for the given application.

The second point is done by a special translator from the code generated by the commercial ESTEREL compiler into FORTH. This translator is tailored for RTX2000. The third point is possible because phases 1 and 2 provide deterministic code and the RTX2000 has a fully predictable behaviour.

In this paper we present the principles of the synchronous approach and the main features of the ESTEREL language. Then, we analyze the advantages and drawbacks of the RTX2000 target architecture. In a third part, we sketch out the code generation process. Finally, quantitative results are given.

2: ESTEREL: a synchronous approach

In this chapter, we present the philosophy of synchronous programming, taking the example of ESTEREL language. The purpose of the synchronous programming is to give an assistance for the conception and the specification of reactive systems[3]. After describing principles of synchronous programming, we present the advantages of ESTEREL for real-time processing.

2.1: The synchronous approach

The synchronous languages address the issue of reactive system programming. Reactivity has been introduced by Harel and Pnueli[4] to designate systems that react to inputs from their environment by sending themselves outputs to that environment. Such systems can be programmed in terms of reaction to activations at logical instants.

The evolution of the system is controlled by external events. The system is fully stimuli dependent, and the reactions to stimuli are atomic.

The synchronous hypothesis amounts to consider ideal reactive systems that react by instantly updating their internal state and producing reactions. This hypothesis is an approximation of the reality but greatly simplifies the study of the application.

The direct and practical consequences is the determinism introduced by this approach. The time is a completely controlled notion and the programming style is simpler and safer.

Languages like ESTEREL[5], SIGNAL[6] and LUSTRE[7], are members of the synchronous language family. The first one has an imperative style and the two others are data flow oriented. In what follows we consider the former.

2.2: The ESTEREL language

ESTEREL is a special purpose language. Designers have introduced concurrency and communication as **first class concept** inside the language. Contrary to many high-level languages, ESTEREL is both parallel and deterministic.

The language is built on a formal mathematical semantics and the ESTEREL compilation gives an efficient finite automaton.

The programming unit in ESTEREL is the module. The declarative part of the module defines the interface between the program and its environment. The body of the module is the imperative part. ESTEREL offers classical constructors: conditional (**if...then...else...end**), iteration (**loop ... end**), composition operators (sequence (**;**), parallel (**||**)). In addition, ESTEREL has an efficient escape mechanism (even working for concurrent processing). The associated statement is (**trap ... end**), and the exception raising is done by an **exit** statement.

Data processing capabilities of the language are deliberately restricted (variables, assignement). They are transfered to the host language through the use of types, functions and procedures.

This simple example of a tank heating shows how to use a **trap** to stop an execution on a signal occurrence (STOP_HEATING). When this signal occurs the temperature value is emitted after a possible incrementation of temperature.

Example:

```
trap HEATING in
        every DEGREE do
                temperature=temperature+1
        end every
        ||
        await STOP_HEATING;
        exit  HEATING
end trap ;
emit TANK_TEMP(temperature)
```

In ESTEREL, communication is supported by signals. The communication is made by instantaneous signal broadcasting using the operator **emit**. Reception is instantaneous and system-wide. All the modules receive the same information at the same instant.

Temporal statements are most important for real-time programming. Time is multiform in ESTEREL and is given by signals. Watch-dogs are implemented with the statement **do ... watching ... end** .

The occurrences of signals are tested by the **await** statement. **present...then...else...end** is another primitive to test the presence of signal in the current instant.

A variant of the previous example, using the do watching statement, gives the preemption priority over the incrementation.

Example:

```
do
        every DEGREE do
                temperature=temperature+1
        end every
watching STOP_HEATING ;
emit TANK_TEMP(temperature)
```

ESTEREL offers other facilities to implement handlers for timeout and exception.

In our example if the STOP_HEATING signal does not occur, the heating cannot stop. So it is wiser to avoid this possibility. In this new specification after one hour the heating stop automatically.

Example:

```
do
        every DEGREE do
                temperature=temperature+1
        end every
watching STOP_HEATING
timeout  HOUR
        emit ERROR
end ;
emit TANK_TEMP(temperature)
```

In this description of the ESTEREL model we have shown some possibilities of this language for real-time specifications. The principle is the synchronous hypothesis that gives a clear and formal semantics to ESTEREL.

ESTEREL makes a total abstraction about physical time, and the most original hypothesis is its "ideal" execution environment. According to this hypothesis, most of the instructions are executed in null time.

Clearly, the transition from ESTEREL to the "real world" raises a problem like the change from the "ESTEREL time" (generally equal to zero) to the instruction duration in the implementation.

Our goal is to minimize the gap between synchronous universe and real world.

A previous experiment has be reported in [8]. A performance evaluation has demonstrated advantages of the use of the both synchronous and asynchronous approaches.

2.3: The ESTEREL compilation

Fig.1 sketches out the compilation process of an ESTEREL program. With respect to our method, the source file and the "OC code" file are the more interesting. The ESTEREL code contains the modules of the application. Modules are basic unit of the ESTEREL programmation and could be included in others (copymodule statement). There may be several instances of the same module in a program.

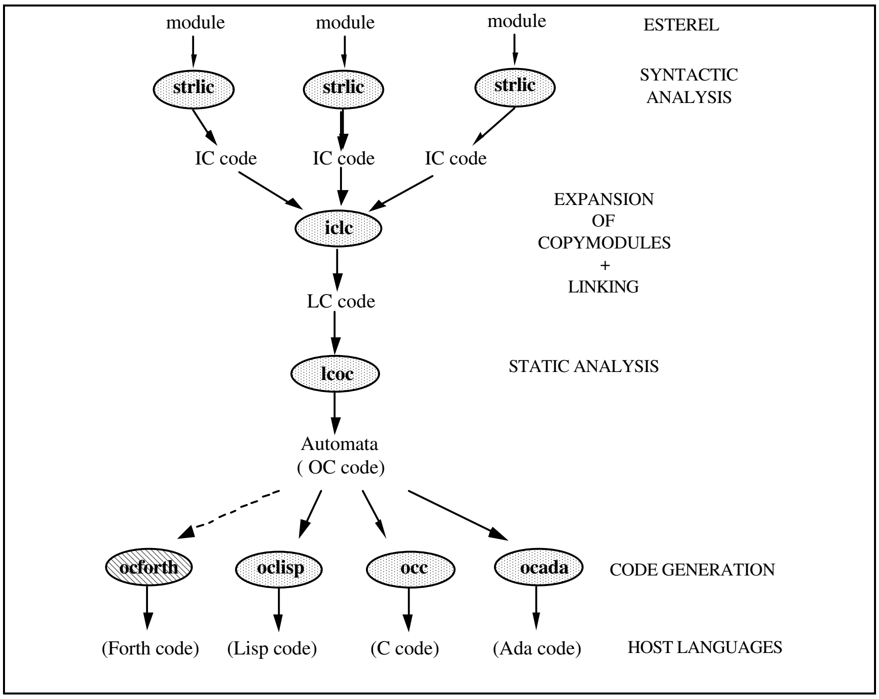

Figure 1: The ESTEREL Compilation Line

After a syntactic analysis ESTEREL code becomes an IC (Intermediate Code) file. Intermediate Code expresses the ESTEREL program into low-level code, and the IC files are linked.

The more complex task for the compiler is producing OC (common code) from IC. The parallelism is sequentialized and fixed point equations for signals are solved. Semantically incorrect programs (non determinism, existence of causality cycle) are rejected at this level, a finite automaton is generated, otherwise.

There are different modes of compilation in ESTEREL ("normal mode" or "cascade mode"). In the first mode, only one automaton is built, whereas in the second mode several automata are generated. The former gives bigger but faster automaton, the latter yields smaller automata whose executions are linked at the run-time.

Remark: ESTEREL and its environment are industrial products[9].

3: The microcontroller RTX2000

RTX2000 is a highly integrated, high-performance microcontroller. We explain in this section the pros and the cons of this architecture for Real-time processing.

3.1: Why a real-time architecture ?

We have chosen RTX2000[10] for its adequacy to real-time application. RTX2000 is a RISC architecture but it does not have the drawbacks of classical RISC. There is no cache memory and no pipe-line instructions so the **synchronization** and the **execution** are **predictable**.

As a RISC architecture the RTX2000 processor runs fast, generally one instruction of the RTX takes **one machine cycle** .

This architecture has **inner-parallelism**, so instructions which do not use the same resource (buses, RAM, timers...) can be combined. For example the return from subroutine can take zero cycle if it is combined with an arithmetic operation.

The extension of the architecture out chip is allowed using an ASIC bus.

We give now some performances of the processor for critical execution. One instruction takes one machine cycle of 100ns. A memory access takes 2 cycles. **Response to interrupt** which is an important feature for real-time control, is four clock cycle latency, and **subroutine call** takes one cycle.

RTX2000 is an autonomous microcontroller which contains the processor plus one hardware multiplier, two stacks, three timers and interfaces to the out-chip memory and the ASIC bus. Memory access is reduced by an intensive use of the two internal stacks.

Because its main functions are executed directly by hardware, RTX2000 is a high-performance controller.

3.2: RTX2000 and real-time applications

RTX2000 is well suited for applications which require high speed execution. The applications range from embedded control, to signal processing, communication or robotics.

RTX2000 is optimized for an integrated real-time control, so there is no complexity inherent to general purpose microcontrollers. The minimal system based on RTX2000 plus external memory can be use as stand alone system.

Besides its architecture, another outstanding feature of RTX2000 is its programming language: FORTH.

FORTH is strongly connected to stack-oriented processing. Its use may be tricky for programmers.

3.3: FORTH: the RTX2000 programming language

3.3.1: FORTH in Real-time application: FORTH is a stack-based high-level programming language. Because of the stack architecture FORTH uses the reverse Polish notation, so that algorithms have to be revisited. For efficiency FORTH is an interesting language but for programming this language is not extremely popular.

Reluctance to FORTH programming style could be an impediment for RTX2000 development.

Fortunally, our method spares the programmer this burden.

The first industrial application of FORTH (1971) was for data acquisition in an Astronomy Observatory. FORTH was introduced as a fast and extensible language able to monitor and/or control physical processes.

FORTH includes rudimentary multitasking facilities[11] (activation and desactivation of tasks and semaphores for communication). This approach allows creation of multiple background tasks that execute asynchronously but it is a low level programming of concurrency.

Attempts to develop kernel for Real-Time Operating systems have been described[12]. Commercial products are anounced by the manufacturer[13].

In our method, concurrency has been resolved at compile time, leading to sequential code easier to execute on the processor.

3.3.2: The FORTH language: The principal entity of FORTH language is the word. This makes FORTH extensible. A word is equivalent to a subroutine or a procedure. It may be composed of machine language instructions or other FORTH words.

A FORTH program is implemented bottom up and is itself a word. A new word definition becomes part of the language.

Here are examples of user's word definitions:

: Open_filling_Valve (output instruction: set a bit on a physical port) … **;**

: Wait_for_fill (activate a timer for x minutes)…**;**

: Close_filling_Valve (output instruction: reset a bit on a physical port) … **;**

: Fill_Tank Open_filling_Valve Wait_for_fill Close_filling_Valve **;**

The words are maintained in a "DICTIONARY" which varies with implementation. The dictionary is a linked list of word definitions. Each entry contains the name of the word, the code to implement this word and a link to the next entry.

FORTH is both a high-level and a low-level language. It is similar to an **assembly language** because of its ability to access to hardware directly. It yields compact and high speed code, it has low overhead subroutine calls and it is efficient for parameter passing.

But it is also similar to **high-level language** because it encourages for structured programming and top-down modular design. It has IF…ELSE…THEN, several loop constructors and a CASE statement. It supports complex data structures.

High-level languages make implicit use of a stack. FORTH requires **explicit** use of the stacks. There are two stacks: the return stack and the parameter stack.

The **Parameter stack** is used to pass parameters for data. The words take arguments from stacks and leave results on stacks.

The **Return Stack** holds subroutine return address, counters of loop iterations, and possibly temporary storage.

4: From ESTEREL to FORTH

The objective is to use a Real-Time language to program an efficient microcontroller. The original language of the microcontroller is not suficiently convenient to describe our application.

In this section we explain how to design application using a synchronous language like ESTEREL and how to implement it on the RTX2000 microcontroller.

This is made, using an automatic translator from ESTEREL to the assembly language FORTH.

4.1: The OC common code

ESTEREL, and other synchronous languages like LUSTRE, can be compiled into finite state automaton. An automaton is coded in a common code (the OC code) which can be translated into different target languages (C, ADA ...).

The OC code contains information about the ESTEREL module, i.e a series of tables relevant to signals, variables, actions, and the automaton. These tables describe all the references to signals and variables and these references are used in the description of actions. The automaton is a decision tree. Actions to be executed. are associated with nodes.of the tree. The OC has a fixed format from which we extract information by lexical and syntactical analyses.

4.2: The OC2FORTH translator

The policy of translation varies according to the ESTEREL compilation mode. If modules are "cascaded", several parts of code must be duplicated (e.g the automaton code in the case of several instances of the same module). In this case, it is worth using arrays to maintain module state variables. On the contrary, for single automaton the best way is to use simple variables.

The principle of the "OC2FORTH" translation is to build the words that describes actions, the variables or data structures which contain information related to the state of the automaton, and words which control automaton evolutions.

So there are two parts in the resulting program:

•1 The "**constant "** part which consists of:

- A word to initialize variables, matrix and vectors,
- Matrix and vector definitions words which contains information about the state of the automaton (automata), addresses of the state words etc...,
- A word to change the state during the automaton evolutions,
- A word to control the transition of the automaton.

•2 The **"variable"** part which is application dependant. It is the result of the information given by the OC file. The actions and the different states of the automaton are translated into words. This part contains also the signal table definitions (signals are manipulated through variables) and the variable and constant tables.

Remark: If external functions or procedures are used, according to the ESTEREL programming style, only the calls are automatically generated. It is the programmer's responsibility to translate their code in the host language (i.e FORTH).

5: Test of performance

In this section we present the outline of a performance analyser for RTX2000 code.

This tool **has been important** to evaluate the effeciency of different translation strategies. Strategy varies according to the choice of size or efficiency criterion (i.e reduction of memory space or reduction of execution time). In the first case we use arrays to store the value of signals and variables. In order to save memory space, signals flags are compacted (bit representation of a boolean). In the second strategy the goal is to reduce execution time. More direct access to data is provided by variables. The access to variables takes two cycles on RTX2000.

The performance analyser **is precious** to verify that time constraints are respected (particulary important for hard Real-time systems).

A FORTH program is a collection of words. A word may use another one, inducing a dependence relationship on words.

The input of the analyser is a disassembled version of the actual RTX2000 code. It proceeds in different steps:

•1 determine, by code inspection, the precedences between words,

•2 evaluate the duration of the different transition of the automaton.

•3 give the results of the shortest and longest execution duration for each words and for each transition.

Step 2 is possible because the duration of each elementary instruction on RTX2000 is known at compile time and because ESTEREL introduces no loops.

In order to illustrate the output of the analyser, we give a table taken from a paper devoted to implementation issues[14]. Durations are expressed in number of cycles. One cycle on RTX2000 takes 100ns.

WORDS	**SPEED STRATEGY Min,Max**	**MEMORY STRATEGY Min,Max**
state0 to state0	15, 15	38, 38
state1 to state1	---, ---	38, 38
state1 to state2	21, 21	44, 44
state2 to state2	54, 92	38,180
state2 to state4	75, 92	167,180
state2 to state3	50, 60	92,104
state3 to state3	28, 38	38, 74
state4 to state4	49, 72	38,127
state4 to state3	46, 73	92,137

The duration may be two times longer in the second solution. On the contrary, the first solution needs more memory than the second (554 vs 783), but the difference would have been greater for a real size example with many input signals.

6: Conclusion

The problem was to give an assistance for the design of Real-Time application. On the one hand, there exist real-time high-level languages, on the other hand, performant devices able to support real-time implementation are available. But there was a gap between the two's.

In the concern for safety and readability, we have chosen ESTEREL as the programming language. It is high-level and deterministic. Into the bargain, ESTEREL has a very precious environment. There exist several code generators, one dedicated symbolic debugger and automaton checkers.

The processor we use is the RTX2000. It allows the efficient implementation real-time applications but its programming language (FORTH) is not widespread.

One **important contribution** of our work has been to make easier the use of RTX2000: no FORTH code has to be written by the programmer. For that, we have designed a compiler from the high level language ESTEREL to the target language FORTH. This compiler is optimized to generate efficient FORTH code.

To be sure that our application respects time constraints (essential for Hard Real-Time systems) we have conceived a tool to evaluate run-time performances for an application.

Because our method is based on a high-level language (ESTEREL), the developement cycle time to system implementation is drastically reduced and safer.

Concerning future extension of this product, we plan to integrate the new ESTEREL statement (**exec**) that allows control of external asynchronous tasks. A second improvement is to make the automatic translation of the transformational part of the application (procedures and functions). At present, the programmer has to make this translation by hand.

7: References

[1] **S.J.Young** *Real-Time Languages: Design and Development* Chichester, Ellis Horwood

[2] Gsi-Tecsi RTC XEC86 *Executive user's guide* Mai 87, Paris, France.

[3] *Special section on: Another Look at Real-Time programming,* **B.Benveniste, G.Berry Eds**, Proceedings of the IEEE, vol. 79, n°.9, September 1991

[4] **D.Harel, A.Pnueli** *On the development of reactive systems* Logic and Models of Concurrent Systems NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Springer Verlag 1985

[5] **G.Berry, G. Gonthier** *Esterel Synchronous Programming Language: Design, Semantics, Implementation,* Rapport de recherche INRIA N° 842, 1988

[6] **B.Benveniste, P.Bournai, T.Gautier, P.Le Guernic** *Signal: A data-flow oriented language for signal processing*, Rapport de recherche INRIA N°378, Mars 1985

[7] **P.Caspi, N.Halbwachs, D. Pilaud J.Plaice** *LUSTRE a declarative language for programming synchronous systems*, Proc. 14th Annual ACM symp. on Principles of Programming languages, Munich, 1987

[8] **C.Andre, L.Fancelli** *A Mixed Implementation of a real-time system,* Amsterdam, North-Holland Euromicro Microprocessing and Microprogramming 30 (1990) 397-402.

[9] *ESTEREL v3: User's manuals,* CISI Ingéniérie, Valbonne, France, 1991

[10] *RTX reference manual,* Harris Semiconductor Corporation June 1988, Melbourne (FL), USA

[11] *PC/FORTH version 3.2,* Laboratory Microsystems, Inc, Marina del Rey (CA), USA

[12] **L.Lindh, F.Stanischewski** *FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel,* Proceedings Euromicro Workshop on Real-Time systems, Université PARIS XI, (p36-40), june 1991

[13] RTX press, vol 1, n°1, Dec 1989, Melbourne (FL), USA

[14] **C.Andre, M.A.Peraldi**, FROM ESTEREL TO FORTH, SIGForth'92 FORTH Language Workshop, Kansas-City USA , March 5-7, 1992