FROM ESTEREL TO FORTH

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
This paper deals with the implementation of a real-time system on a high-performance microcontroller whose programming language is FORTH (Harris RTX2000). We have apply the new synchronous programming style (ESTEREL language) to the design. We aim to combine the efficiency of FORTH with the reactive and real-time high-level constructs of ESTEREL.
Automatic translation from ESTEREL to FORTH, optimization strategies, and timing performances are presented. A simple example illustrates the full process.

A. INTRODUCTION

Our application domain is the control and the command of industrial processes. We apply the new synchronous programming style [IEEE91] to the design of smart microcontrollers. Our programming language is ESTEREL [ESTE91], a synchronous and imperative language suitable for real-time application. ESTEREL can be compiled into general purpose languages (C, ADA,…) and its compilation gives a deterministic and finite state automaton.
Our target processor is the Harris RTX2000 [RTX88], a high performance microcontroller which is well suited for real-time control. RTX2000 has the speed characteristic of a RISC architecture but it has not the drawback of other RISC processors. Its execution is fully predictable (no cache memory and no pipeline). RTX2000 programming language is FORTH.
FORTH is a semi compiled language. So, it has the speed characteristic of a compiled language and the advantages of an interpreted language. FORTH can also provide direct accesses to hardware.

The description of our applications requires a reactive language.
FORTH is extensible; it could be possible to implement reactive mechanisms but they are not primitives of the language. On the contrary, ESTEREL is intrinsically reactive, so we have chosen to take the advantages of both these languages and we have implemented an ESTEREL to FORTH translator in order to combine high-level design with fast execution.
To test the efficiency of the FORTH code produced, we have developed a performance analyser. We know at compile time how many cycles take all the automaton transition. This tool is particularly important for hard real-time systems which have to respect critical temporal constraints.
In this paper, we sketch out the principles of ESTEREL compilation, then we present the RTX2000 architecture and its characteristics. In a third part, we analyze the ESTEREL to FORTH translator, different strategies of translation are suggested. Finally we give the

principles of the performance analyser and some quantitative results.

B. INTRODUCTION TO ESTEREL

ESTEREL is a language based on the synchronous approach. The synchronous languages address the issue of reactive system programming. Reactivity has been introduced by Harel and Pnueli [HaPn85] to designate systems that react to inputs from their environment by sending themselves outputs to that environment.
ESTEREL is an imperative and synchronous language. It has **concurrency and communication as first class concepts inside the language**. Contrary to many high-level languages, ESTEREL is both parallel and deterministic.
The general features of ESTEREL are:

- High-level language suitable for control-oriented programming,
- Communication supported by signals (instantaneous broadcasting),
- Concurrency (parallel operator),
- Temporal statements (watch-dogs...)

Temporal instructions are used to handle (multiform) time and synchronization. Because of the synchronous hypothesis the following statements have precise meanings:

- `every 100 Ms do emit tenth_of_second end`
- `await 50 Metre; emit finished`

If Ms is an input signal occurring every "physical" millisecond then the first statement provides a 10 Hz clock. The output signal tenth_of_second, is emitted for the first time at the very same time as the 100 th occurrence of Ms., then it is emitted every 100 occurrences of Ms, accurately.
The second statement emits the signal "finished" at the instant when Metre (a non standard form of time (!) generated by some length gauge) occurs for the 50 th time.

The following example is a foretaste of the ESTEREL programming style. We want to monitor the speed of a truck. First we program a chronometer. Every Milliseconds (`Ms`) a counter is incremented. On every occurrence of "`HowLong`", the reactive box emits the duration (number of Ms) occurred since the previous occurrence of `HowLong`.

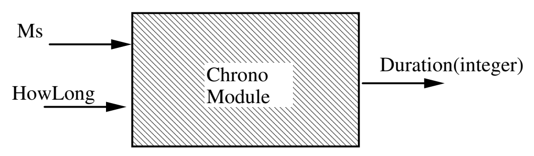

Figure 1: ESTEREL example

ESTEREL module for truck time-keeping:

```
module CHRONO:
  %Interface declaration
input HowLong, Ms;
output Duration(integer);
  %Body of the module
var counter : integer in
%counter is a local variable
  loop
    counter:=0;
    trap COUNTING in
      every Ms do
        counter := counter+1;
      end every
      ||
      await HowLong; exit COUNTING;
    end trap;
    emit Duration(counter)
    %Signal generation
  end loop
end var
end module
```

Remark:
"`%`" starts a comment till the end of the line.

Now, in order to control the speed of a truck, we include the CHRONO module and we implement the control part of the speed. A possible ESTEREL program is:

```
module SPEED_CONTROL:
```

```
input Ms,Metre,Terminated,FaultFixed;
output TruckArrived;
constant Min_Duration: integer;

signal Duration(integer), HowLong in
%Local signals
[
  do
    trap MOVE in
      every Metre do
        emit Howlong;
        %request for duration
        if ?Duration > Min_Duration
        % optional then: do nothing
        % because no violation
        else exit MOVE
          % Exception raising
        end if
      end every
    handle MOVE do
    %handler for speed limit violation
      …
      %Recovery process
      …
      await FaultFixed;
    end trap
  watching Terminated
  % When the control is here, the
  % vehicle has reached its target
  emit TruckArrived;
]
||% Parallel operator
  copymodule CHRONO
  % Elsewhere defined module
end signal
end module
```

The ESTEREL code contains the modules of the application. Modules are basic units of the ESTEREL programming and could be included in others. They may be several instances of the same module in a program.
Suppose you have to control the speed of three independent trucks in a critical part of a railway, concurrently. Then, take three instances of the SPEED_CONTROL module. Signals, except Ms are renamed in module instances. The resultant module is:

```
module MULTI_SPEED:
input Ms, Metre1, Metre2, Metre3,
  Terminated1, Terminated2,Terminated3;
output FaultFixed1, TruckArrived1,
  FaultFixed2, TruckArrived2,
  FaultFixed3, TruckArrived3;

  copymodule SPEED_CONTROL [signal
      Terminated1/Terminated,
      Metre1/Metre,
      FaultFixed1/FaultFixed;
        constant 1000/Min_Duration]
||
  copymodule SPEED_CONTROL [signal
      Terminated2/Terminated,
      Metre2/Metre,
      FaultFixed2/FaultFixed;
        constant 2000/Min_Duration]
||
  copymodule SPEED_CONTROL [signal
      Terminated3/Terminated,
      Metre3/Metre,
      FaultFixed3/FaultFixed;
        constant 1500/Min_Duration]
end module
```

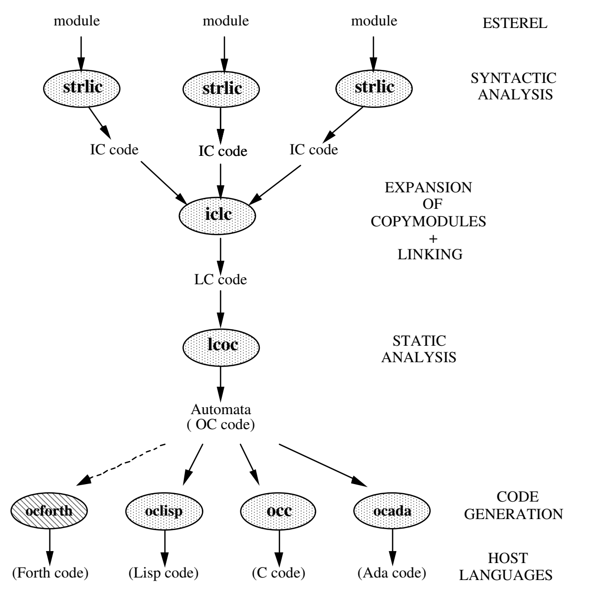

Figure 2: The ESTEREL Compilation Line

The compilation of an ESTEREL program gives a finite state automaton. Fig.2 outlines the compilation process of an ESTEREL program.

After a syntactic analysis ESTEREL code becomes an Intermediate Code. Intermediate Code expresses the ESTEREL program in low-level code. Next, 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 must be solved. Semantically incorrect programs (either non deterministic or with causality cycle) are rejected at this level. A **finite deterministic automaton is generated**, otherwise.

Only the OC file is significant in what follows.
Fig.3 is the automaton semantically equivalent to the Speed control module. Annotations associated with arcs are conditions/actions. The determinism is ensured by evaluating conditions in a fixed order (small numbers by the head of the arcs).

C. RTX2000: A PROMISING MICRO-CONTROLLER

1 RTX2000 ARCHITECTURE

RTX2000 is a high-performance microcontroller. We explain here the advantages of its architecture for our class of applications.
RTX2000 integrates neither pipeline nor cache memory, so that its **execution is predictable**. Its low interrupt latency allows **fast response to event**. Using the potentials of the architecture (parallelism, hardwired instruction, high-speed math, …), outstanding performances can be achieved.
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 (Parameter stack and Return stack).
The **parameter stack** is requested for memory access, data storage…. The **return stack** stores subroutine return addresses. These stacks can be useful for **multitasking management**. In this case, task context switching could be reduced to changing the stack pointer. Attempts to develop kernel for Real-Time Operating systems had been described [LinSta91]. Commercial products had been announced by the manufacturer [RTX89].

2 RTX2000 ENVIRONMENT

The RTXDS [RTXSP89] is a PC-based set of integrated tools useful for software development on RTX2000. This system is made up of two parts:

- the"host" part, which runs on PC (editor)
- the "target" part is based on a RTX2000 card and provides an interactive execution environment (debugger).

3 RTX PROGRAMMING LANGUAGE

Beside its architecture, another noteworthy feature of the RTX2000 is its programming language: FORTH. More exactly, RTX2000 has its own instruction set designed to execute the FORTH language directly in hardware.
FORTH is strongly connected to stack-oriented processing. It has both characteristics of **high-level programming** language:

- Structured programming,
- Extensible language,
- Support for data structures,

and of **assembly language**

- Interrupt management
- Compact code
- Low overhead for subroutine.

FORTH includes multitasking facilities [PCF89] (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.

RTX2000 executes directly FORTH. The memory can be divided into RAM and ROM, which is be useful for embedded systems. In the speed control example we use this possibility, the access mechanism to data structure is stored in ROM whereas data are stored in RAM.

FORTH has a powerful mechanism to append new words in a "dictionary". So-called "defining words" are useful to define arrays and structures. Their behaviour is different at run time and at compile time.
Example:

```
\ Matrix of n bytes, dimension L × C
{ : MATRIX_BYTEn } ( L C n --)
  VARIABLE * * ALLOT
  DOES> ( L C Cmax n --)
    >R ( L C Cmax --)
    ROT SWAP ( C L Cmax--)
    * + R< * ( add--) ;
```

On RTX2000, at compile time ALLOT reserves space in RAM and a pointer to the matrix is stored in ROM. At run time the DOES> statement pushes the address of the matrix onto the stack and a cell is accessed by indirect addressing. Another capability that has been exploited is the **"vectored execution"**. It provides a mechanism for executing a word whose address is stored on the top of the stack. We have applied the principle of the vectored execution to the automaton. The tick (**'** or **[']**) statement gives access to the word address.

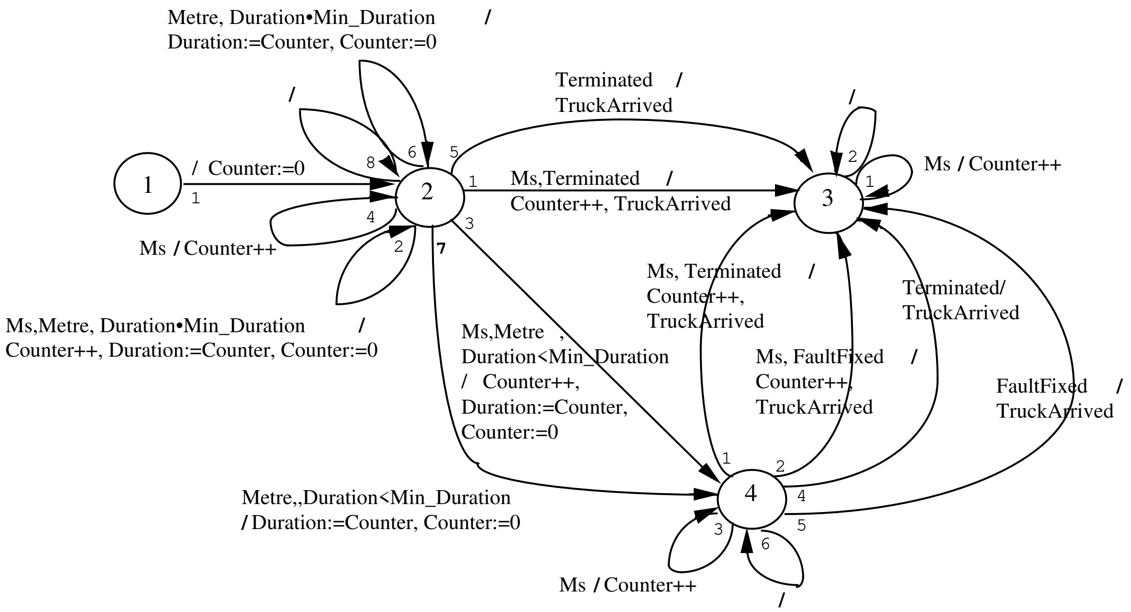

Figure 3: The SPEED_CONTROL automaton

```
signals: 9
0: input: Ms 1 pure: bool:0 % previous: first:% %lc: 2 7 0%
1: input: Metre 2 pure: bool:1 % previous: 0% %lc: 2 11 0%
2: input: Terminated 3 pure: bool:2 % previous: 1% %lc: 2 18 0%
3: input: FaultFixed 4 pure: bool:3 % previous: 2% %lc: 2 30 0%
4: output: TruckArrived 5 pure: % previous: 3% %lc: 3 8 0%
5: local: single: 4  %name: Duration previous: 4% %lc: 4 8 0%
6: local: pure:  %name: HowLong previous: 5% %lc: 4 27 0%
7: exception: pure:  %name: MOVE% %lc: 8 7 0%
8: exception: pure:  %name: COUNTING% %lc: 8 7 1%
end:
```

TABLE 1: The OC signal table of SPEED_CONTROL module

```
actions: 11
0: goto: %lc: 1 1 0%
1: present:0 %lc: 2 7 0%
2: present:1 %lc: 2 11 0%
3: present:2 %lc: 2 18 0%
4: present:3 %lc: 2 30 0%
5: output:4 %lc: 3 8 0%
6: if:$9 (4,#1000) %lc: 11 4 0%
7: reset:5 %lc: 5 5 1%
8: call:$1 (5) (#0) %lc: 7 2 1%
9: call:$1 (5) ($13 (5,#1)) %lc: 9 15 1%
10:call:$1 (4) (5) %lc: 13 2 1%
end:
```

TABLE 2: The OC action table of SPEED_CONTROL module

Example:
```
' STATE0  (in interactive mode )
['] STATE0 (in compilation mode)
```

When the address is on the top of the stack, the keyword **EXECUTE** performs the code of the state (i.e executes a reaction).
Another advantage of RTX2000 is the possibility to execute upto three different instructions at the same time.
We have made the most of all these possibilities in our translator (see the next section).

D. THE ESTEREL TO FORTH TRANSLATOR

1 THE OC CODE

The OC code is the source file for our translator. The OC file contains all the significant information to generate the FORTH code of the application. This file is composed of a series of tables (signal table, procedure table, type table, action table…) and an automaton. Successive indirections are necessary for accessing to information.
Hand-coding would be error-prone and tedious so, we have developed an automatic OC to FORTH translator using YACC.

2 PRINCIPLE OF THE EXECUTION

The execution of the automaton has to be controlled. A reaction of this automaton consists in different steps:
- Built the current event
- Activate the automaton,
- Execute the transition.

The first two steps are the responsibility of the environment. A description of the execution machine can be seen in [AnMaPa91]. The third step generates the actual reaction.
The first step builds the current event. This event gives an image of the environment and is made up of all the signals set during this phase.
Each input signal is a structure with two fields:
- flag: boolean
- value : T

where flag denotes the presence of the signal, value is its current value, and T is its type. On the occurrence of a signal S, the environment must assert S.flag and set S.value (if defined).
The second step is the activation of the automaton by a special triggering event.
An automaton can be seen as a set of DAGs (Direct Acyclic Graph). A DAG is associated with each state. Labels on the edges of DAGs are either tests or actions. Actions may have either internal or external (output signals) effects. The execution of a transition (third step) corresponds to an execution path on the DAG associated with the current state. When a transition terminates (the next state is reached) all the input flags are reset.
The system runs as indicated in Fig.4.
In our FORTH implementation, a (FORTH) word is associated with each phase, see annotation between brackets on Fig.4.

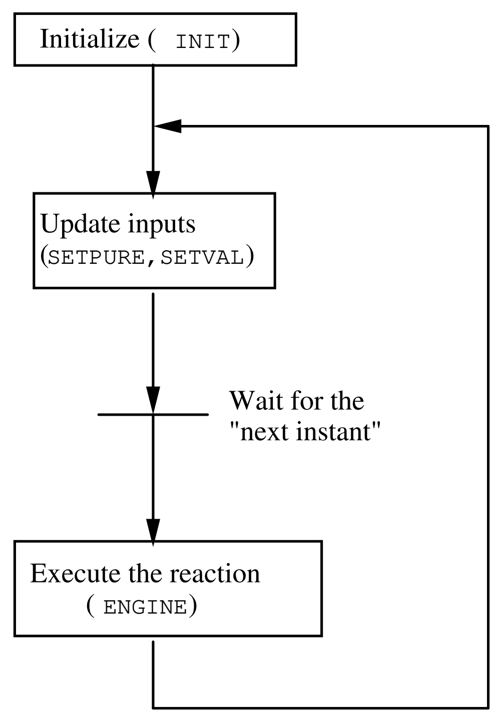

Figure 4: FORTH execution algorithm

The word `INIT` initializes data and signals. The signals are updated by the environment (`SETPURE` and `SETVAL` words). The `ENGINE` word controls the evolution of the automaton and resets signals after the completion of the transition.
Further details about the implementation are given in section D.3.

3 THE OC TO FORTH TRANSLATION

This translation can be executed according to different optimization criteria (reduction of execution time or reduction of memory space). In this section we give the translation of the OC tables for the different cases.

a. Signal, variable and action translation

In the case of **memory saving criteria** we use arrays to store the value of signals and variables. Each signal is translated into a constant which represents its index in the array of signal values. In order to save memory space, signal flags are compacted (bit representation of a boolean) in a data structure named `VECT_SIGPRESENT_<module name>`.

Example:

OC signal table of SPEED_CONTROL module (extract from the oc file): see Table 1.

FORTH translation of signal table (memory saving criterion)

```
( 1.1 DEFINITION OF SIGNALS )

0 CONSTANT Ms_V0
1 CONSTANT Metre_V1
2 CONSTANT Terminated_V2
3 CONSTANT FaultFixed_V3
0 CONSTANT TruckArrived_V4
4 CONSTANT Duration_V4
VARIABLE counter_V5
10 CONSTANT Min_Duration

\ Definition of constants and
\ arrays for signal management

1 CONSTANT NB_INSTANCES_SPEED_CONTROL
4 CONSTANT NB_SIGIN_SPEED_CONTROL
1 CONSTANT NB_SIGOUT_SPEED_CONTROL

\ Definition of array for output and
\ input signal value

NB_INSTANCES_SPEED_CONTROL
     NB_SIGOUT_SPEED_CONTROL 2
     MATRIX_BYTEn
     MAT_SIGOUTVAL_SPEED_CONTROL

NB_INSTANCES_SPEED_CONTROL
     NB_SIGIN_SPEED_CONTROL 2
     MATRIX_BYTEn
     MAT_SIGINVAL_SPEED_CONTROL
```

There might be several instances of the same module in ESTEREL. If so, signals are differentiated by indexes.
The packed representation of flags may spread over many bytes. So, a special access mechanism has been developed. Of course, this kind of access penalises timing performances, all the more that it is often called.
For the sake of simplicity, we present here solutions for systems not exceeding 16 input signals. The following example is a translation of the actions of the automaton.
Example:

OC actions table of SPEED_CONTROL module: (extract from the oc file): see Table 2.

FORTH translation of action table (memory saving criteria)

```
: BIT2NUM DUP 0 = IF DROP 1
     ELSE 1 SWAP -1 FOR 2* NEXT THEN ;

( 2 ACTIONS )
( 2.1 JUMP ACTION )
 : A1_SPEED_CONTROL FALSE ;

( 2.2 PRESENT SIGNAL TESTS )
 : A2_SPEED_CONTROL
  DUP Ms_V0 BIT2NUM AND 0> ;
 : A3_SPEED_CONTROL
  DUP Metre_V1 BIT2NUM AND 0> ;
 : A4_SPEED_CONTROL
  DUP Terminated_V2 BIT2NUM AND 0> ;
 : A5_SPEED_CONTROL
  DUP FaultFixed_V3 BIT2NUM AND 0> ;

( 2.3 OUTPUT ACTIONS )
```

```
 : A6_SPEED_CONTROL
   " OUTPUT:TruckArrived_V4 " CR ;

( 2.4 ASSIGNMENTS )
 : A7_SPEED_CONTROL 0 counter_V5 ! ;
 : A8_SPEED_CONTROL
   counter_V5 @ 1 + counter_V5 ! ;
 : A9_SPEED_CONTROL
   counter_V5 @ Duration_V4
   MAT_SIGINVAL_SPEED_CONTROL ! ;
( 2.5 CONDITIONS )
 : A10_SPEED_CONTROL Duration_V4
   MAT_SIGOUTVAL_SPEED_CONTROL @
   Min_Duration > ;
```

In the second strategy, the goal is to **reduce execution time**, at the expense of memory saving. More direct access to data is provided. A valued input signal is represented by two variables (one for the flag, one for the value). The access to variable takes two cycles.
The same OC code translated in FORTH with execution time criterion is:

FORTH translation of signal and action table (speed execution criterion)

```
1 CONSTANT NB_INSTANCES_SPEED_CONTROL
( 1.1 DEFINITION OF SIGNALS )

VARIABLE Ms_V0
VARIABLE Metre_V1
VARIABLE Terminated_V2
VARIABLE FaultFixed_V3
VARIABLE TruckArrived_V4
VARIABLE Duration_V4
VARIABLE counter_V5
10 CONSTANT Min_Duration

( 2 ACTIONS )
( 2.1 JUMP ACTION )
: A1_SPEED_CONTROL FALSE ;

( 2.2 PRESENT SIGNAL TESTS )
: A2_SPEED_CONTROL Ms_V0 @  0> ;
: A3_SPEED_CONTROL Metre_V1 @ 0> ;
: A4_SPEED_CONTROL
  Terminated_V2 @ 0> ;
: A5_SPEED_CONTROL
  FaultFixed_V3 @ 0> ;

( 2.3 OUTPUT ACTIONS )
 : A6_SPEED_CONTROL .
   " OUTPUT:TruckArrived_V4 " CR ;
( 2.4 ASSIGNMENTS )
 : A7_SPEED_CONTROL 0 counter_V5 ! ;
 : A8_SPEED_CONTROL
   counter_V5 @ 1 + counter_V5 ! ;
 : A9_SPEED_CONTROL
   counter_V5 @ Duration_V4 ! ;

( 2.5 CONDITIONS )
 : A10_SPEED_CONTROL
   Duration_V4 @ Min_Duration > ;
```

In section E we give some results concerning the time performances of each words in the different cases of translation.

The implementation of procedures and functions is not supported by ESTEREL but by the host language. Therefore, words are generated but their body is under user's responsibility.

b. Automaton translation

The automaton is constituted of several states. Each state is translated into a word which contains a series of presence signal test and the associated actions. We use the IF… ELSE…THEN statement to implement the decision tree of the automaton.
Example: the second state of `SPEED_CONTROL` automaton.

```
: STATE2_SPEED_CONTROL
A2_SPEED_CONTROL IF
   A8_SPEED_CONTROL
   A4_SPEED_CONTROL IF
      A6_SPEED_CONTROL
      3 CHANGESTATE_SPEED_CONTROL EXIT
   THEN
   A3_SPEED_CONTROL IF
      A9_SPEED_CONTROL
      A7_SPEED_CONTROL
      A10_SPEED_CONTROL IF
         2 CHANGESTATE_SPEED_CONTROL EXIT
      THEN
      4 CHANGESTATE_SPEED_CONTROL EXIT
   THEN
   2 CHANGESTATE_SPEED_CONTROL EXIT
THEN
A4_SPEED_CONTROL IF
   A6_SPEED_CONTROL
   3 CHANGESTATE_SPEED_CONTROL EXIT
THEN
A3_SPEED_CONTROL IF
```

```
   A9_SPEED_CONTROL A7_SPEED_CONTROL
   A10_SPEED_CONTROL IF
      2 CHANGESTATE_SPEED_CONTROL EXIT
   THEN
   4 CHANGESTATE_SPEED_CONTROL EXIT
THEN
2 CHANGESTATE_SPEED_CONTROL ;
```

At the end of each transition the variable which contains the current state of the automaton is updated by the `CHANGESTATE_SPEED_CONTROL` word:

```
: CHANGESTATE_SPEED_CONTROL
   (Auto_Inst_Nb N°state--) SWAP
   VECT_CURRENTSTATE_SPEED_CONTROL ! ;
```

c. The execution machine

The execution machine is a set of FORTH words. An array (`VECT_STATES_SPEED_<modulename>`) plays a central role, its entries are addresses of words associated with states.

```
4 CONSTANT NB_STATES_SPEED_CONTROL

NB_STATES_SPEED_CONTROL VECTOR_BYTE2
     VECT_STATES_SPEED_CONTROL
```

Like for data structures, some parts of the execution machine vary according to the strategy.

Execution machine with memory saving strategy

```
: INIT
   \ State Addresses are stored in
   \ the state array
   ['] STATE0_SPEED_CONTROL
      0 VECT_STATES_SPEED_CONTROL !
   ['] STATE1_SPEED_CONTROL
      1 VECT_STATES_SPEED_CONTROL !
   ['] STATE2_SPEED_CONTROL
      2 VECT_STATES_SPEED_CONTROL !
   ['] STATE3_SPEED_CONTROL
      3 VECT_STATES_SPEED_CONTROL !
   \ Initialization of the current
   state
   \ array and the signal array
   NB_INSTANCES_SPEED_CONTROL -1 FOR
    1 I VECT_CURRENTSTATE_SPEED_CONTROL
   !
    0 I VECT_SIGPRESENTS_SPEED_CONTROL
   !
   NEXT ;
```

```
: SETPURE (N°signal Auto_Inst_Nb --)
   VECT_SIGPRESENTS_SPEED_CONTROL SWAP
   (addSP N°signal--)
   DUP 0 = IF
      DROP 1 ( 1--)
   ELSE (addSP N°signal--)
     1 SWAP 1- ( I 1 N°signal-1--)
     FOR 2* NEXT (  n--)
   THEN
   OVER @ ( addSP n valSP--)
   OR SWAP ! ;

: SETVAL
   (Val N°signal Auto_Inst_Nb --)
   OVER OVER (Val N°signal Auto_Inst_Nb
         N°signal Auto_Inst_Nb --)
   SETPURE
   (Val N°signal Auto_Inst_Nb--)
   SWAP MAT_SIGINVAL_SPEED_CONTROL  (Val
addsignal--)
   ! ;

: ENGINE
\ For all the instances of the
\ automaton: execute the automaton
   NB_INSTANCES_SPEED_CONTROL -1 FOR
     I VECT_CURRENTSTATE_SPEED_CONTROL
     @ VECT_STATES_SPEED_CONTROL @
     I SWAP EXECUTE
     \ The signal are reset after each
     \ transition
     I VECT_SIGPRESENT_SPEED_CONTROL !
   NEXT ;
```

Execution machine with speed execution strategy

```
: INIT
   \ State Addresses are stored in
   \ the state array
   ['] STATE0_SPEED_CONTROL
     0 VECT_STATES_SPEED_CONTROL !
   ['] STATE1_SPEED_CONTROL
     1 VECT_STATES_SPEED_CONTROL !
   ['] STATE2_SPEED_CONTROL
     2 VECT_STATES_SPEED_CONTROL !
   ['] STATE3_SPEED_CONTROL
     3 VECT_STATES_SPEED_CONTROL !
   \ Initialization of the current
   \ state array
   NB_INSTANCES_SPEED_CONTROL -1 FOR
     1 I
     VECT_CURRENTSTATE_SPEED_CONTROL !
     \ The boolean of input signals are
     \ reset one by one
     0 Ms_V0 !
     0 Metre_V1 !
     0 Terminated_V2 !
     0 FaultFixed_V3 !
```

```
   NEXT;

: SETPURE ( Addbool--) 1 SWAP ! ;

: SETVAL ( Val Addsignal Addbool--)
   1 SWAP ! (Val Addsignal--) ! (--) ;
: ENGINE
   \ For all the instances of the
   \ automaton: execute the state of
   \ the automaton
   NB_INSTANCES_SPEED_CONTROL -1 FOR
      I VECT_CURRENTSTATE_SPEED_CONTROL
      @ VECT_STATES_SPEED_CONTROL
      @ EXECUTE
      \ The boolean of input signals are
      \ reset one by one
      0 Ms_V0 !
      0 Metre_V1 !
      0 Terminated_V2 !
      0 FaultFixed_V3 !
   NEXT ;
```

E. PERFORMANCE ANALYSER

The analyser gives quantitative results, so that, the efficiency of the different implementations can be compared. Moreover, the analyser is used to guarantee that the implementation meets its temporal constraints.

1 PRINCIPLE OF THE EVALUATION

Simulations techniques are not satisfactory for real time systems. When possible, a static verification is better. In our approach, three factors contribute to a performance evaluation at compile time. Those three factors are RTX2000, FORTH and ESTEREL.
Each (elementary) instruction in RTX2000 has a fixed duration (most are one clock cycle, some two clock cycles). This outstanding feature is due to the absence of cache memory and pipelining.
A FORTH program is a collection of words. This structure greatly simplifies the analysis of the generated code. When implemented on RTX2000, upper level FORTH's words are expanded by the compiler (e.g: DO, LOOP, …). These "utilitary" words are easily localized. Their non recursive use can be checked and their duration evaluated. Our FORTH engine adds some "management" words. We have define only non recursive words. Furthermore, all our iterations are fully controlled: the number of iterations is known at compile time.
Therefore, the application independent part of our implementation has a predictable behaviour. The application dependent part is the translation of the tables and the automaton described in the OC file generated by the ESTEREL compiler. Each state of the automaton is associated with a word (see Par D.3). It is a representation of a DAG (hence, loop free and recursion free).

The analyser works as follows:

1st Disassemble the RTX program. (Parsing phase);

2nd By code inspection, determine the precedences (a word uses another word) and apply a topological sorting (Sorting phase);

3rd Evaluate the duration (shortest and longest execution) for each word taken in the previous order (Longest and shortest path algorithms).

2 QUANTITATIVE RESULTS

In Table 3, execution times are expressed in number of cycles. One cycle on RTX2000 takes 100 ns. Words are listed in an order such that a word never call a word below. Dashed lines stand for unused words.
The nine last lines in Table 3 are significant results for timing performances. 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 real size examples with many input signals.

F. CONCLUSION

FORTH was created in the late 1960's specifically to solve a real time control problem. Since, the complexity of the applications has been ever increasing. A way to deal with this complexity is to program systems with languages in which concurrency and communication are first class concepts. So is ESTEREL, an imperative synchronous language. In this paper, we have advocated a design method programming in ESTEREL and implementing in FORTH, well-known for its efficiency.

The principal impediment in the use of synchronous languages is the gap between the synchronous idealized world, and the asynchronous real world. It is an implementation issue to prove that a realization meets the synchronous hypothesis.

WORDS	SPEED STRATEGY Min,Max	MEMORY STRATEGY Min,Max
= :	3, 4	3, 4
* :	3, 3	3, 3
MATRIX_BYTE2 :	13, 13	13, 13
VECTOR_BYTE1 :	1, 1	1, 1
VECTOR_BYTE2 :	2, 2	2, 2
VECT_STATES_SPEED_CONTROL :	6, 6	6, 6
VECT_SIGPRESENT_SPEED_CONTROL:	---,---	6, 6
VECT_CURRENTSTATE_SPEED_CONTROL:	5, 5	5, 5
MAT_SIGOUTVAL_SPEED_CONTROL :	---,---	17, 17
MAT_SIGINVAL_SPEED_CONTROL :	---,---	17, 17
MAT_SENSVAL_SPEED_CONTROL :	---,---	17, 17
MAT_SENSBOOL_SPEED_CONTROL :	---,---	17, 17
> :	4, 4	4, 4
CHANGESTATE_SPEED_CONTROL :	12, 12	12, 12
A1_SPEED_CONTROL :	1, 1	1, 1
A6_SPEED_CONTROL :	8, 8	8, 8
A7_SPEED_CONTROL :	5, 5	5, 5
A8_SPEED_CONTROL :	9, 9	9, 9
A9_SPEED_CONTROL :	8, 8	25, 25
< :	3, 4	3, 4
I :	4, 4	4, 4
MATRIX_BYTE1 :	13, 13	13, 13
SIGNAUX_SPEED_CONTROL :	---,---	19, 19
A10_SPEED_CONTROL :	10, 11	27, 28
INIT :	101,101	101,101
ENGINE :	26, 28	49, 49
BIT2NUM :	---,---	10, 14
A2_SPEED_CONTROL :	11, 11	21, 25
A3_SPEED_CONTROL :	11, 11	21, 25
A4_SPEED_CONTROL :	11, 11	21, 25
A5_SPEED_CONTROL :	11, 11	21, 25
SETPURE :	4, 4	24, 26
SETVAL :	6, 6	48, 50
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

TABLE 3: Temporal performances

We have considered the implementation issue of an ESTEREL program on a Harris RTX2000 microcontroller. Since the programming language of the RTX2000 is FORTH, the central problem is to **provide a correct and efficient translation from ESTEREL to FORTH**.
Static correctness results from ESTEREL compilation followed by an automatic translation. However, it is not sufficient for hard real-time systems that requires predictable temporally constrained behaviours. We ensure dynamic correctness by a performance analyser. Two implementation strategies and their validation have been presented in this paper.

The "OC to FORTH" translator can be easily adapted to other optimizations. Even if further experiments are necessary, we believe that our method offers a real possibility of combining high-level description with efficient implementation. for controller design.

G. REFERENCES

[AnMaPa91]: **C.Andre, J.P.Marmorat, J.P.Paris** *Execution machine for ESTEREL,* ECC91, Grenoble France 1991, pp 1672-1677 Vol2 HERMES Ed

[ESTE91]: *ESTEREL v3: User's manuals* , CISI Ingéniérie, Valbonne, France, 1991

[FORTH83]:**W.P.Salman,O.Tisserand,B.Toulout** *FORTH* EYROLLES edition,(1983)

[HaPn85]: **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

[IEEE91]: *Special section on: Another Look at Real-Time programming,* A.Benveniste, G.Berry Eds, Proceedings of the IEEE, vol. 79, n°. 9, Sept 1991, (pp 1268-1336)

[LinSta91]: **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, (pp36-40), june 1991

[PCF89]: PC/FORTH version 3.2 Laboratory Microsystems, Inc, Marina del Rey (CA), USA.

[RTX88]: RTX reference manual, Harris Semiconductor Corporation June 1988, Melbourne (FL), USA

[RTX89]: RTX press, vol. 1, n°.1, Dec 1989, Melbourne (FL), USA

[RTXSP89]: RTX software product, Harris Semiconductor Corporation April 1989, Melbourne (FL), USA