Hard Real-Time System Implementation on a Microcontroller

Charles ANDRE Marie-Agnès PERALDI

Départ. LASSY, I3S - Université de Nice-Sophia Antipolis / CNRS
Bd Napoléon III, F 06041 NICE Cedex, FRANCE

Abstract: This paper addresses the issue of the design of reactive applications subject to hard temporal constraints. We advocate the use of the synchronous language ESTEREL for the design of such applications The target machine is a real-time microcontroller (RTX2000 from Harris).

This paper deals with the automatic translation from an ESTEREL program down to microcontroller code, and the validation (both logical and temporal) of the implementation.

Keywords: Real-time systems, Hard real-time systems, Implementation, Verification.

1 Introduction

Controllers must react to events (stimuli) coming from the environment by producing responses (outputs). This behaviour is typical of reactive systems [1]. When a reaction must strictly occur within a priori given time frames the systems are said to be hard real-time systems.

A most precious resource to manage is time. Real-time systems require a predictable behaviour in terms of guaranteed deadlines. An other crucial requirement is safety, since failures of a real-time system could cause a disaster.

The design and the implementation of hard real-time systems raise several problems.

First, we have to express the expected behaviour of a system involving concurrency, communication, synchronization, and exception handling. Extra requirements like temporal constraints must be taken into account.

The second problem concerns the choice of a target processor. Fast, and above all, predictable processors seem to be adequate.

The third issue is to ensure that the implementation meets the specifications. Both the temporal and logical correctness of the application must be proved.

This paper presents our solutions to those problems.

A synchronous formalism [2,3] (the ESTEREL language) is used to express the expected behaviour in a convenient and rigourous way. However, the synchronous automaton produced by the ESTEREL compiler must be interfaced with the asynchronous real world. Interfacing is taken in charge by an execution machine whose responsability is to collect the events from the environment and to execute transitions of the automaton. The synchronous hypothesis greatly simplifies the design but this strong simplification of reality must be criticized and checked at the implementation level.

We have chosen the RTX2000 processor (from HARRIS) as hardware support for implementation. This microcontroller is fast in execution and its behaviour is predictable. Due to its hardware structure and its speed this processor is closed to the requirements of real-time systems.

Combining ESTEREL and RTX2000 allows verifications at different levels of the conception.

First, ESTEREL is based on a formal mathematical semantics. The logical correctness of the application can be checked using tools of the ESTEREL environment. The proofs are done on the actual object code obeying the "WYPIWYE" principle (What You Prove Is What You Execute).

Second, our automatic translation from ESTEREL programs into RTX2000 code preserves this logical correctness.

Temporal correctness is established at the implementation level by checking the RTX2000 code. Realistic bounds on system reactions are deduced from the actual code.

This paper is dedicated to the conception of real-time microcontroller using this approach. First we pointed out the characteristics of the ESTEREL language and the RTX2000 processor. We describe the execution machine that interfaces the reactive kernel with the environment. Then, we give the principle of the automatic translation from ESTEREL to RTX2000 (OC2FORTH translator). Finally, we give a method to evaluate the timing performance of the implementation.

2 Programming: the ESTEREL language.

a. ESTEREL

ESTEREL is a synchronous language. The synchronous hypothesis amounts to consider systems that react by instantly updating their internal state and producing reactions. This hypothesis may be not realistic but greatly simplifies the dezsign of application. ESTEREL [4] is an imperative language dedicated to the programming of reactive application.

First, as ESTEREL is a high-level language, aprograms are understandable, easy to maintain and to debug.

Second, this language is built on a formal mathematical semantics. The compilation of an ESTEREL program yields a deterministic and finite state automaton.

Third, thanks to dedicated tools available in the software environment of ESTEREL [5,6], the logical correctness of the program can be carried out by finite state machine analyses.

b. Language statements

ESTEREL is a *special purpose language* more concerned with control than in data processing. The programming unit of ESTEREL is the module. The declaration part specifies the interface between the module and its environment, whereas the body of the module defines its expected behaviour in an imperative style.

Classical constructors like conditional test (**if…then…else…end**) and iteration (**loop…end**) are available.

Contrarily to many other high-level languages ESTEREL is both *parallel* and *deterministic*. The parallel operator is (||)), and the sequential one is (;). ESTEREL has a powerful *exception* handling treatment with the escape mechanism (**trap…end**), the exception raising is done by the **exit** statement.

Communication is supported by instantaneous broadcasting of signals, all the modules receive the same signal at the same instant. An ESTEREL program generates signals by the **emit** statement.

Physical time is unknown to ESTEREL. *Temporal statements* deal with logical multiform time provided by input signals. Because of the synchronous hypothesis, the sequence of statements:

```
await 2 MS; await 4 MS;
```

is strictly equivalent to

```
await 6 MS;
```

Where MS is some input signal that might be interpreted as physical millisecond. Few real-time languages allow such an equivalence.

The primitive **present…then…else…end** test the presence of a signal in the current instant.

The **do…watching** statement is a most significant contribution to timeout and watch-dog programming.

3 The execution machine for ESTEREL.

An ESTEREL program describes the synchronization skeleton of an application. This program can be compiled into a semantically equivalent automaton. As a mathematical object, this automaton reacts instantaneously to input events (stimuli) by producing outputs (reactions).

Actual implementation of the automaton introduces delays. Moreover, incoming events occur asynchronously. Those two facts give rise to a **gap** between the model and the implementation. In order to reduce this gap, we have conceived an execution machine [7] which interfaces the asynchronous real world with the synchronous reactive kernel. To carry out a reaction, the execution machine proceeds in successive steps:

- build the current event for the automaton,
- activate the automaton for its transition,
- build the output event after execution of the reaction.

During the first step the execution machine collects signals from the environment, and build the current event for the automaton. This step could be quite different according to the strategy of collection. Whatever the strategy, it is mandatory not to lose signals.

The activation of the automaton is the second step. It is the execution machine responsibility to start the current instant and to present the event. Internal automaton variables are changed and output functions are called.

During the last step, output signals are collected and an "output event" is emitted. The automaton returns the control to the machine and the current instant ends.

In what follows, we explain how to implement the execution machine on the RTX2000.

4 Implementation: FORTH on RTX2000

a. The RTX2000 microcontroller

The RTX2000 [8] is a high performance and autonomous microcontroller dedicated to the design of embedded real-time applications. Because of the potential of the architecture (parallelism, Asic buses, hardwired instructions…), outstanding performances can be achieved. As a RISC architecture, the RTX2000 runs fast but it has neither pipeline nor cache memory so that its behaviour is predictable.

Some interesting features deserve to be mentioned. A subroutine call takes 1 clock cycle, memory access takes 2 cycles and interrupt latency is 4 clock cycles. A clock cycle on the processor is 100 ns.

RTX is a stack-based architecture, the chip has a *parameter stack* for manipulation of data, memory access and multitasking management and a *return stack* for subroutine calls.

The environment for designing RTX2000 applications consists in a FORTH environment running on a PC, and a dedicated development system RTXDS [8] providing interactive debugging.

b. The RTX2000 programming language: FORTH.

RTX2000 executes directly the FORTH language. FORTH is a stack-based programming language. Because of its stack structure FORTH uses the reverse polish notation.

FORTH programs are designed top-down but are typically implemented bottom-up. The "word" is the basic element of FORTH. A word is equivalent to a procedure which takes its parameters and leaves results on a stack.

Words defined in an application are maintained in a dictionary and become part of the language. This makes FORTH flexible and extensible.

FORTH has both characteristics of a low-level and a high-level language. It is very efficient but it is not extremely popular among designers. Like an assembly language, it can manage interrupt, it yields a compact and efficient code and, due to the words structure, it gives a low overhead for subroutine calls.

5 From program to implementation

Our method proposes to combine the advantages of the ESTEREL language and the RTX2000 processor. Hand-coding ESTEREL into RTX2000 programs would be difficult, tedious and error-prone. We have conceived an automatic translator from ESTEREL into FORTH that preserves the determinism and the semantics.

We have integrated the hardware characteristics of the processor for code generation. So, different options are available. The code can be optimized either for minimal execution time, or.for minimal memory use.

An application generated by our OC2FORTH translator contains two main parts.

The first part is variable and application dependant. Informations for the translation are extracted from the OC file generated by the ESTEREL compiler.

The second part is invariant and implements the execution machine.

Knowing the formal grammar of the OC code and using YACC, we have developped our translator. The OC file contains a series of tables (signal table, procedure table, action table…) and the automaton. Accesses to information is done by successive indirections. OC2FORTH translates signals into variables, and actions into words.

The automaton is a collection of states and transitions (change in states).. With each state is associated a word that expresses the different transitions from this state. A "state" word can be thought of as a Direct Acyclic Graph (DAG). The nodes of this DAG are signal presence tests. The edges are labelled by actions to be executed. Leaves are "goto" statements which indicate the next state.

The execution machine is composed of a set of FORTH words which manipulate data structures. The execution machine runs through different steps Fig.1.

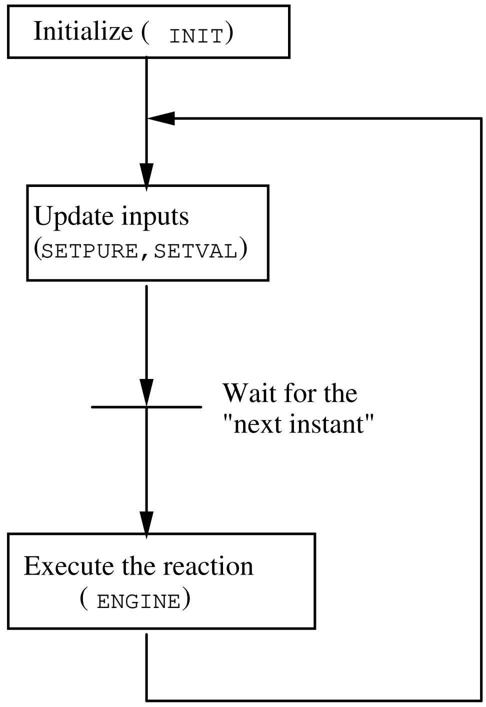

Figure 1: FORTH execution algorithm

The first step (INIT word) initializes the control structures and the variables of the system. The second step (SETPUR, SETVAL) corresponds to the collection of the inputs and the construction of the current event to be presented to the automaton. The last step (ENGINE) is the execution of the automaton. Details about the different techniques of translation are given in [10].

6 Performance evaluation

In this part, we explain how to compute, at compile time, the duration of any reaction.

This possibility results from a conjonction of favourable factors: the code generated by the ESTEREL compiler, the very nature of the implementation language (FORTH) and the processor itself (RTX2000).

Our program has been automatically generated by a sequence of transformations.

- i) a compilation (from ESTEREL to OC),
- ii) a translation (from OC to T-FORTH),
- iii) a compilation (from T-FORTH to RTX code)
- iv) and a disassembling (from RTX code to the code to be analyzed).

We have very little influence on code production during phases iii and iv. Phase i is a bit more versatile (directive given to the ESTEREL compiler). On the contrary, phase ii offers a great opportunity to produce fully mastered programs. In order to control the execution of the automaton and the accesses to data, we have predefined several Forth's words (see Para.5).

We aim at producing predictable code. So, we have discarded recursion and undefined iterations. The ESTEREL compiler generates purely sequential and linear code, i.e. the execution of a reaction in OC is free of iteration or recursion. Our translation does not introduce recursion. Some iterations have been useful for data initializations. Only definite loops (do loops or for-next loops) with the number of repetitions known at compile-time have been used. Reactions are directly translated from the OC automaton, therefore they are loop-free.

The FORTH words are the only entities of the original program that can be easily identified in the disassembled code. The whole program is itself a word. Most of the words have been defined by the programmer. Some words have been generated by the compiler. The code produced by the FORTH compiler is such that each word W has one and only one entry label. A word has at least one EXIT instruction. Thus, words are the natural unit for analysis. Our method is based on the evaluation of the execution of a word.

RTX2000 lends itself to this kind of evaluation because its execution is fully predictable. Though it is a RISC architecture, it has no instruction "pipeline" or cache to flush when performing branches or calls. The duration of each transition is fixed and known.

Executing a word is executing a sequence of instructions: its duration is the sum of the duration of each instruction. It is out of the question to compute all possible executions (exponential algorithms). Instead, we have chosen to compute classes of executions for which we can give bounds on duration. The complexity of this approach is linear in the number of instructions.

The method proceeds as follows:

- 1) Topological sorting on words
- 2) For each word W taken in the topological order do:
  - 21) Associate a rooted directed graph with W,
  - 22) Unfold possible loops,
  - 23) On the resulting dag compute longest and shortest paths.

Comments:

1. A word can call another one, inducing a precedence relationship. Calls are easily found by code inspection. Topological sorting leads to an ordered list of words such that a word in the list never calls a word farther in this list.
2. Taking words according to this ordering ensures that any word called has been already processed.

21. We have a rooted graph because a word has one and only one entry. Vertices of the graph are addresses of instructions. Edges are labelled by instructions and are directed. An execution is represented by a path on this graph.
22. In some words (initialization words) there may exist loops easily detected by a depth-first transversal of the graph. Since those loops are of restricted types and with a known number of repetitions, we can erase the back-edges and unfold the loops. Of course, the do not actually unfold a loop, but we note that the duration of the loop is n times the duration of its body, where n is the number of repetitions.

After the loop eliminations, we get a dag.

23. The edges of the dag are weighted by a duration. For an elementary instruction the duration is a known integer value. For a call to a word, the duration has been given by a previous computation on this word. In this case the duration is a couple of integers (a minimal and a maximal duration). Weights associated with edges in a loop body are multiplied by the number of iterations done.

Then, the minimal and the maximal durations for a word are deduced by applying to an acyclic directed graph, a search for the shortest and the longest paths from the root down to the terminal vertices.

When all the words have been analyzed, we have bounds on the duration of each word. Satisfaction to hard temporal constraints can then be checked.

A question arises: are those bounds closed to the upper lower bound and the lower upper bound?. According to our experimentations the answer is positive, and mainly for the duration of "state words" which are the most important words for the in-line computations (generation of reactions). The reason is that those words are loop-free and result from a linear execution encoding.

A full description of the method requires a semantical background beyond the scope of this paper. Further details are available in a technical report [11].

7 Conclusion

The problem was the design of hard real-time controllers. There exist real-time languages and special purpose architectures to support such implementation but both reveal different degrees of inadequacy to meet hard real-time requirements like safety and predictability.

We have chosen a synchronous formalism (ESTEREL) for the design. Because it is deterministic and high-level, ESTEREL seems to be adequate to express real-time and reactive constraints. Into the bargain, it has a precious environment for debugging and for checking the logical correctness.

The target microcontroller is RTX2000 from Harris. As a RISC architecture the RTX2000 runs fast but its behaviour remains predictable.

An important contribution of our work has been to give an automatic tool for safe an efficient translation from ESTEREL code into RTX2000 programming language. This automatic translation preserves the logical correctness of the application. Temporal verification is done at the implementation level. A performance analyzer checks the RTX2000 cod at compile-time. It associates a realistic interval of duration with each reaction. This phase is fundamental to assert whether the synchronous hypothesis is satisfactory for our application or not, and thereby, if hard temporal constraints are met. This static analysis is possible because of the predictable behaviour of the processor and the determinism of the code generated by the ESTEREL compiler.

In this paper we have considered almost pure reactive systems. These systems carry out little data processing. An extension of ESTEREL dealing with asynchronous tasks is now available (introduction of the EXEC statement [7]). Execution machine dedicated to this new version of the language are under development.

8 References

[1] 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

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

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

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

[5] R.de Simone, D. Vergamini *Aboard AUTO* Tecnical Report INRIA N°111, October 1989

[6] G. Boudol, R.de Simone, D. Vergamini *Expêriment with Auto and Autograph on a simple case of sliding window protocol,* Tecnical Report INRIA N°83, March 1987

[7] C.Andre, J.P.Marmorat, J.P.Paris, *An execution machines for ESTEREL,* ECC'91, Grenoble FRANCE, July 1991, PP1672-1677, Vol.2 HERMES Ed,7

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

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

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

[11] C.Andre, M.A.Peraldi, COMPUTATION OF THE EXECUTION DURATION OF A RTX2000 CODE, Technical Report, I3S, Sophia-Antipolis, France , May, 1992