Esterel Programming in FMS

C. ANDRÉ*, S. ELKHATTABI**, J.C. GENTINA**, M.A. PÉRALDI*
(*) Laboratoire Informatique, Signaux, Systèmes (I3S)
Université de Nice-Sophia Antipolis / CNRS
(**) Laboratoire d'Automatique et d'Informatique Industrielle de Lille (LAIL)
Ecole Centrale de Lille / CNRS

Abstract

Lowest levels of FMS are reactive real-time systems. Timeliness, safety and dependability are major requirements for those systems. Classical programming techniques bring only partial solution. We propose a "synchronous approach" based on the ESTEREL *language that makes specification, programming and verification easier* [1].

Keywords: FMS, Synchronous programming, Monitoring, Control.

1 Introduction

Reactive Real-Time Systems (RTS, for short) are systems that maintain permanent interactions with their environment (reactive feature) and are subject to various constraints (real-time requirements). The term real-time can be misleading. Even if they are important, *temporal constraints* are not necessarily the main requirements for RTS. *Safety* and *dependability* must be mastered too, and often those requirements are more critical than timeliness.

According to this definition of RTS, *Flexible Manufacturing Systems* (FMS) are typically RTS. The lowest levels (coordination and local control) are tightly connected to their environment. The constraints are the harshest at those levels. Such systems rely heavily on computing devices (Programmable Controllers, Numerically Controlled Machines, ..., micro and minicomputers).

In order to ensure efficiency, safety and dependability, high-level quality software must be provided. The widely spread approach, that uses tasks (written in some sequential language) managed by a real-time executive, is not satisfactory with respect to the quality of the generated software. Modern programming languages (like ADA), that integrate high-level data, control and programming constructs, seem more suitable for complex applications and maintainable solutions. But, because they are general-purpose, those languages are better in transformational than in reactive system programming.

Recently, the synchronous approach [1] has been developped to address the issue of RTS programming. ESTEREL [2] is a high-level imperative synchronous language well-suited for discrete-event systems whose behaviour is driven by event occurrences. ESTEREL makes the programming of such systems easier, and it leads to efficient implementations. Moreover, because of its ability to abstract complex behaviours, ESTEREL can be used as a model for the expected behaviours.

In this paper, we advocate the use of ESTEREL in modelling and programming of the lowest levels of FMS. With the help of a simple example, we explain how ESTEREL may improve the design of dependable local controllers. Modular approach to control, detection, diagnosis and recovery is made easier. The goal of this paper is to illustrate the adequacy of the main ESTEREL's constructs to the control issues. The role of ESTEREL is not restricted to high-level and generic descriptions of expected behaviour. Because of its simple and sound mathematical semantics, ESTEREL allows verifications. Event-driven simulators are available. For critical applications a formal verification of the logical correctness of program can be done.

2 F.M.S as reactive real-time systems

2.1 Layered approach to FMS

FMS are complex to design. To reduce the complexity, a hierarchical decomposition of these systems into layers is adopted by designers [3, 4, 5]. Main layers are *supervision*, *coordination* and *local control*. The flexibility of the system is integrated at the supervision and the coordination levels, whereas reactivity and real-time constraints characterize the local control.

[1]This research was in part supported by the "Direction des Recherches et des Etudes Doctorales" (DRED)

The control flows through the different levels down to the process level. Obviously an upward flow must exist, at least to confirm the execution of orders. This upward flow is associated with the *monitoring* of the application. It is necessary to ensure safety and dependability of the system. When something goes wrong, the fault should be detected right away, its consequences should be kept local, and if possible, the fault should be fixed. Layers facilitate fault detection, fault containment and fault recovery. Thus, we have to program both normal and abnormal control. Exceptions and interruptions handlings are absolute necessity for programming such systems.

2.2 Our synchronous approach

Petri Nets are the model generally used to describe the coordination of actions at the higher levels. At the lower level, where the real-time constaints are the harshest, (classical) Petri nets are helpless because of their absence of preemptive statements. *Safety aspects* and *failure detection* of the physical process should be integrated at this level. The methods currently available for designing Fms lack systematic approach to the integration of dependability in systems [6].

The reactive feature of control, fault detection and possible recovery demands reactive models. Obviously, general purpose languages are not dedicated to this kind of programming. We have chosen a *synchronous approach* to the design of Fms. This approach greatly simplifies the design because of its power of abstraction: systems are abstracted to discrete event systems, time is abstracted to a logical one, and communication is supposed to be instantaneous. Grafcet (Function Charts for Control Systems) [7] is a well-known *synchronous model*. Recently introduced synchronous formalisms (Statecharts) and *synchronous languages* (Esterel, Lustre and Signal) allow more complex reactive specifications. A special issue of the proceedings of the Ieee [1] has been devoted to synchronous programming languages. These languages are based on a formal mathematical semantics and this property allows to make proofs on programs. Esterel is an appealing solution to control system programming, that is the reason why we have chosen it for the design of safety aspects of Fms. This language is briefly presented in the next section.

2.3 The synchronous language Esterel

Esterel [2, 8] is a synchronous language composed of a few basic instructions. The language is based on a simple and sound mathematical semantics. The programming unit of Esterel is the *module*. The declarative part of a module specifies the interface between the module and its environment.

The Esterel programming style is *imperative*. The language includes some classical constructs such as iterations and conditionals, but also reactive statements dedicated to control. Esterel has powerful statements to express *communication, concurrency, temporal behaviour* and *preemption*.

- Communication is supported by signals. Signals are *broadcast* system-wide. This communication mode is powerful because all the modules have the same information at the same instant. Signals are generated by the "*emit*" statement.
- Concurrency: Instructions and modules can be composed in parallel. The parallel operator in Esterel is $||$. It has the particularity of not introducing non-determinism in programs.
- Temporal statements: Esterel considers a logical and multiform time. "Time" is associated with signals and can represent physical time, metre, or litre as well. The statement "*present . . . then . . . else . . . end*" tests the presence of a signal at the current instant. The "*await . . .*" statement waits for an event occurrence and therefore is useful for triggering events.
- Exception and preemption are very common in real-time systems. Esterel offers two basic constructs: "*do . . . watching . . . timeout . . . end*" and "*trap . . . in . . . handle . . . end*" statements. We will see in section 3.2 the expression of a "multi-level" watchdog in Esterel.

A noteworthy feature of the Esterel compiler is that concurrency and reactive statements are compiled into a "flat representation", i.e. a *deterministic finite state machine*.

The "*perfect synchrony hypothesis*" [2] assumes that reactions are instantaneous. This hypothesis entails that all classical statements and internal control take no time. When specifying temporal behaviours with Esterel, we adopt a *discrete event system* point of view. Changes in time are expressed by event occurrences. Reactive statements, like "`await ...`", allow waiting for future occurrences, and by the way, waiting for future instants.

3 Programming local control in Esterel

3.1 Actions

A frequently encountered situation in F.M.S. control is the request for doing some action and then waiting for its completion. Of course, the duration of the action is not zero. The piece of program "`emit DoAction; await ActionDone`" will do nicely. Note that, a special statement has been introduced in ESTEREL to express asynchronous executions, i.e. "`exec Action(<reference list>) (<expression list>)`". Concurrent sequences of actions are easily specified by ESTEREL programs. Such programs are very simple and they do not take advantage of the language capabilities.

3.2 Controlled Actions

Generally, actions are temporally constrained: when the order to execute is given, the action has to terminate before some deadline. In order to program a temporally constrained action, it is sufficient to run the control module and a detection module concurrently: "`exec Action || run Timewatch`". `Timewatch` is a generic module for temporal monitoring.

Specification of temporal guards

- event "`DoAction`" (`DoA`, for short): order to execute the action,
- event "`InitialStateLeft`" (short: `ISL`): occurs when the execution effectively starts,
- event "`FinalStateReached`" (short: `FSR`): occurs when the action is completed successfully.

In ESTEREL, events are represented by "*signals*".

Requirements

- date(`ISL`) - date(`DoA`) $\leq$ `INERTIA` units of time
- date(`FSR`) - date(`ISL`) $\leq$ `DEADLINE` units of time

Esterel module for a generic Temporal Guard

```
module Timewatching:
input DoA, ISL, FSR, Top;
output Error: integer, Done;
constant INERTIA, DEADLINE: integer;
constant Stuck, Lost: integer;
 await immediate DoA;
 do
  await INERTIA Top;
  emit Error(Stuck)
 watching ISL
 timeout
  do
   await DEADLINE Top;
   emit Error(Lost)
  watching FSR
  timeout
   emit Done
  end
 end
end module
```

Of course, constants and effective signals must be passed to the module as arguments. Detection of other types of error can be done in an analoguous way by reactive modules. The following example illustrates the case of sensor failure.

4 Example

4.1 Operative part

We consider a Safety Access Sytem (SAS) for a lathe (Fig 1). The door is magnetically driven by a magnetized-piston. Limit switches (Hi1, Hi2, Lo) indicate the state of the door (opened or closed). The jack has two limit switches (J-in, J-out).

Figure 1: Safety Access to a lathe

4.2 Specification

Orders given to the SAS, are "`Open`" and "`Close`". Malfunctions may be caused by actuator or sensor failures. Possible failures must be detected. For safety purpose, orders are filtered: in case of inconsistency, commands may be not applied. Of course, warning messages must be emitted.

First, we have to list the actions and the sensors. For each action we list sensors to be set to "1" and sensors to be set to "0" at the successful completion of the action. Temporal parameters must be given, too.

Action	Inertia	Deadline	Set 1	Set 0
Open	InO	DlO	Hi1,Hi2	Lo
Close	InC	DlC	Lo	Hi1,Hi2

`InO, InC, DlO, DlC` are integers.

4.3 Signals

Sensors yield the status of the physical process. This information is used to generate input signals. The signal generation can be done either at the Esterel program level (systematic polling of the sensors), or in the input handlers of the execution machine. The latter is more efficient.

For each action `A` we define

$$\begin{aligned} S1A &= \text{conjunction of sensors set to 1} \\ S0A &= \text{disjunction of sensors set to 0} \end{aligned}$$

In SAS, signals associated with opening and closing actions are:

$$\begin{aligned} S1O &= Hi1 \bullet Hi2 \\ S0O &= Lo \\ S1C &= Lo \\ S0C &= Hi1 + Hi2 \end{aligned}$$

Derived signals are useful for error detection:

$$NS0O = not\ \ S0O\ and\ NS0C = not\ \ S0C.$$

Normal termination of closing (opening, resp.) is defined by `isC` (`isO`, resp.) :

$$isC = \overline{S1O} \bullet S0O \bullet S1C \bullet \overline{S0C}$$

$$isO = \overline{S1C} \bullet S0C \bullet S1O \bullet \overline{S0O}$$

whereas *sensor fault* is characterized by the following combination of signals:

$$SensorFault = S1O \bullet S0O + S1C \bullet S0C$$

$$+S0C \bullet \overline{S1O} + S0O \bullet \overline{S1C}$$

4.4 Control + Detection

Different detections can be added to the nominal control of the SAS. Fig 2 shows the skeleton of the Esterel program that expresses the closing with fault detection. The program is made of a set of modules embedded in nested `trap` statements. A trap has priority over any nested trap. Statement "`run M`" acts as an inclusion of the module M , written elsewhere. Copy can be done with renamings. For instance the instanciation of `Timewatching` module (Fig 2) is given parameters for the closing action.

Figure 2: Esterel program for closing

If the normal execution terminates successfully, then the control is preempted from the trap `DONE` and an `Ack` signal is emitted to the upper level. In case of sensor failure or timeout detection, a `Nack` signal is emitted with an error indicator. Thanks to the synchronous hypothesis, `Ack` is emitted at the very instant when the action terminates, and `Nack` is issued as soon as a fault is detected.

The module for opening the door is similar.

4.5 Improved control

The previous solution is too simplified. First, opening and closing are not independent actions. Second, the detection of failures is important, but trying to recover from a faulty situation is more interesting. In this section we suggest some improvements to the programming of the SAS control.

Basic idea

The program for "safe control" is built with *communicating reactive modules*. For FMS applications we have chosen restricted forms of modules. A *reactive box* (Fig 3) has one input signal "`Do`" and two output signals "`Ack`" and "`Nack`". The latter is valued by integers. "`Do`" triggers an action, "`Ack`" is emitted at the successful completion of this action, “Nack” is emitted in case of failure.

Figure 3: Reactive box

Example: jack control

The hydraulic jack may breakdown. Two actions are available: “$\mathtt{J}^+$” and “$\mathtt{J}^-$”. Two sensors are associated with the jack: “J-in” and “J-out”. Inertia and deadline can be given for the motion of the piston. Therefore, we have two reactive boxes (one for the backward and one for the forward motion of the piston). Either acknowledgement, or negative acknowledgement can be sent back by those boxes to the SAS box, which is at the first upper level.

Including recovery

Some other detections can be done. Particularly, if there is an *intrusion* while closing, a safety action must be taken and an immediate re-opening of the SAS must be performed. In order to have a systematic detection of errors and a recovery from these errors, we add a Diagnosis and a Recovery module to the initial skeleton. Module gains in “intelligence” and they can be used in a recursive way [12].

```
module SafeAction:
input DoA, ... ;
output Ack, Nack:integer, ... ;
...
trap RUNNING in
 run Filtering
 ||
 trap ABNORMAL in
  run Control
  ||
  run Detection
  ||
  run Diagnosis
 handle ABNORMAL do
  run Recovery
 end trap
handle RUNNING do
 emit Ack
end trap
```

The Filter ensures the consistency of actions by filtering some of them [13]. The Detection module includes detection of timeout, intrusion, sensor faults and generates signal with an error level. The Diagnosis computes this information and reroutes the control for a Recovery from these errors or exit the control from the current level with the emission of a Nack signal. The Recovery treatment can be either a “try again” of the action, or a “execute an alternative action”.

5 Logical correctness

Esterel provides the users with smart software environment for simulation and program proving.

5.1 Simulation

Xsimul [9] is a graphical extension of the interactive simulator of Esterel. This environment is a window-based tool running under X11. Such tool is very useful for checking the logical correctness of programs. Main sequences and abnormal situations can be applied to the simulator in order to observe the behaviour of the system.

5.2 Proofs

Some other tools are available for the verification and the proof of safety properties. Auto [10] is a system designed at manipulation inside a formal calculus. It constructs finite automaton from relevant terms and then allows “projection” according to observational criterion. An Esterel program, once compiled, is amenable to this treatment. Results of this operation can be visualized with Autograph [11], the graphical interface of Auto.

6 Conclusion

Face to the ever-increasing complexity of industrial control systems, a hierarchical approach to the design is generally adapted. The lower layers of the hierarchy are *real-time reactive systems*. Their design has been the subject of this paper.

In order to improve the dependability of systems, layers are composed of modules endowed with some “intelligence”. A module can act as a filter with respect to the control. Moreover, it is able to detect failures and even to fix them. The control, the fault detection and the fault recovery are cooperating activities. According to the Cartesian approach, these activities should be described separately and then composed. In this paper we have advocated the use of a synchronous language "ESTEREL" in the design of the lower layers of industrial control systems.

Languages are suitable for expressing expected behaviours. ESTEREL is well-equipped to specify reactive statements in a precise and concise way. As a synchronous language, ESTEREL composes very well. Contrary to asynchronous languages, the ESTEREL compiler generates an efficient code which is *predictable* and *with low overhead at run-time*. Last, but not least, the *logical correctness* of synchronous programs can be easily checked.

The example presented in this paper points out the power of the ESTEREL stuctures and statements to deal with communication, exception and concurrency. The modules implemented in this example are very standard ones. They can be considered as instances of some generic modules. This observation leads us to developping a library of generic re-usable components [12, 13].

References

[1] A. Benveniste and G. Berry., "Another look at real-time programming," *Proceeding of the IEEE*, vol. 79, pp. 1268–1336, September 1991.

[2] F. Boussinot and R. De Simone, "The ESTEREL language," *Proceeding of the IEEE*, vol. 79, pp. 1293–1304, September 1991.

[3] M. Combacau and M. Courvoisier, "A hierarchical and modular structure for F.M.S. control and monitoring," First Internationnal Conference on A.I., Simulation and Planning in High Autonomy Systems, Tucson, Arizona, March 1990.

[4] J. Bourey, E. Castelain, J. Gentina, and M. Kapusta, "CASPAIM: A computer aided design of the control system of FMS," pp. 131–135, IMACS Annuals on Computing and Applied Mathematics, 1989.

[5] H. Panetto, *Une contribution au génie automatique: Le prototypage des machines et systèmes automatisés de production.* Master's thesis, Université de Nancy I, Janvier 1991.

[6] S. Elkhattabi, D. Corbeel, and J.-C. Gentina, "Integration of dependability in the conception of FMS," pp. 249–255, 7th IFAC Symposium on Information Control Problems in Manufacturing Technology, Toronto (Canada), May 1992.

[7] International Electrotechnical Commission, "Preparation of function charts for control systems," Tech. Rep. 848, IEC, 1988.

[8] "ESTEREL V3: User's manuals," Tech. Rep., CISI Ingénierie, Valbonne (France), 1991.

[9] "ESTEREL C simulation manual," Tech. Rep., CISI Ingénierie, Valbonne (France), 1988.

[10] R. de Simone and D. Vergamini, "Aboard AUTO," Tech. Rep. 111, INRIA, October 1989.

[11] G. Boudol, R. De Simone, and D. Vergamini, "Experiment with AUTO and AUTOGRAPH on a simple case of sliding window protocol," Tech. Rep. 83, INRIA, March 1987.

[12] M.-A. Péraldi, *Conception et Réalisation de Sytèmes Temps-Réel par une Approche Synchrone.* Master's thesis, Université de Nice-Sophia Antipolis, Juillet 1993.

[13] S. Elkhattabi, *Intégration de la surveillance de bas niveau dans la conception des systèmes à événements discrets: Application aux systèmes de production flexibles.* Master's thesis, Université des Sciences et Technologies de Lille, Septembre 1993.