EXECUTION MACHINES FOR 'ESTEREL'

C ANDRE I3S-LASSY, CNRS - Université de Nice-Sophia Antipolis,
J.P MARMORAT , J.P PARIS CMA, ENSMP, Sophia Antipolis

is a synchronous language very convenient for the specification and the programming of reactive systems. Its compile utomaton equivalent to the program. Several industrial applications have demonstrated the feasibility of ESTEREL tions. However, a unified approach to the integration of the ESTEREL automaton needs to be developed.
is an attempt to define an abstract execution machine for ESTEREL. Roughly speaking, one needs to interface a us world (the real world) to a "synchronous" automaton. The abstract execution machine is proposed for that purpose problems induced by the new statement 'exec', which takes account of non instantaneous actions, are also analyzed.

Synchronous language, execution machine, reactive system, real-time system.

I. INTRODUCTION

A reactive system [HaPn85] can be thought of as a black box that reacts to input events by generating output events. This kind of system is permanently interacting with its environment. The moments, when the system reacts, are called "**instants**". What the system has to do, at a given instant, may be specified in behavioural terms. **Synchronous formalisms**, such as the language 'ESTEREL', are suited to this purpose [Dzier90].

Most of the "**real-time" applications** are reactive, because of the direct influence of the environment on their evolutions. Impact of synchronous languages on real-time system programming has been studied in a recent paper [Berry1-89]. Some applications are so reactive that they can be fully programmed by a synchronous language: e.g the digital wrist watch expressed in ESTEREL [Berry2-89], in Statechart [Harel87], ... Other real-time applications may include significant parts of computations that are usually programmed by general purpose languages: e.g in robotics [CosFav90], automatic control [AndFan90], ... In this case, the synchronous language describes a

is briefly presented; it is of great import to many industrial applications.

The second part is devoted to **the abstract execution machine**. Since this machine is abstract, no reference is made to any specific operating system. Only cooperating components associated with well-defined functions are described.

The last part considers **two particular realizations**. The first one is based on a **UNIX** system, and as such is easily portable. The second one, uses a real-time multitasking system (a **SCEPTRE**-like executive [BNI84]) and is convenient for small embedded systems.

II. EXECUTION OF A REACTION

II.1 ESTEREL OVERVIEW

ESTEREL is an **imperative synchronous language** born from a joint INRIA-ENSMP project. An industrial compiler for ESTEREL v3 is available [Cisi90]. In this paper, the reader is assumed to know fundamentals of syntax and semantics of this language.

Example of ESTEREL program: "A watering system controller"

The system is composed of an electrical valve (V) and a humidity sensor (H) which yields the degree of humidity of the soil (an integer value from 0 (dry) to 100 (soaked)). The watering operation (W) is called with an argument giving the degree of humidity that should not be exceeded. The value of H is sampled every minute (MINUTE signal). The watering stops when the desired degree of humidity is reached. The duration of the watering, expressed in minutes, is returned as a result.

An ESTEREL program may be:

```
module WATERING:

input MINUTE, W (integer);
%W argument is the degree of humidity
output      open_VALVE,
            close_VALVE, DURATION
(integer);
sensor H (integer); %humidity sensor

var REFERENCE, COUNTER:integer in
        await W; %wait for the order
        REFERENCE:= ?W;
        % reference to be reached
        COUNTER:=0; % counter for the
duration
        emit open_VALVE;
        trap SPRAY in
                every MINUTE do
                        if REFERENCE>?H
                                % sampling H
                        then exit SPRAY
                                % SPRAY over
                        else
COUNTER:=COUNTER+1
                                % increment
duration
                        end %if
                end %every
        end; %trap SPRAY
        emit close_VALVE;
        emit DURATION(COUNTER)
end %var
.
```

II.2 RESULTS OF THE COMPILATION

The compiler yields one (several in some cases) automaton(s), variables of different types and external references, for actions.

With each input signal, a pair <flag, value> is associated. Let S be an input signal. The flag component, S.flag, is a boolean which is true if and only if the signal S has occurred at the "current instant". This flag is reset when the instant ends. The value component, S.value, if defined, is a remanent information, which is updated on every occurrence of S.

A sensor is associated with a value component that is imposed by the environment. With respect to the automaton, a sensor is a read-only variable.

Internal variables may be either explicitly declared in the program, or created by the compiler for internal uses (e.g: counters of occurrences).

The automaton acts on the environment through output functions. The simplest functions are restricted to an assignment to some external variable (e.g a physical output port).

The transitions of the automaton are guarded by predicates over input signal and sensor values.

The automaton generated for "Watering" is as follows:

WATERING Automaton

```
State 0
        goto 1

State 1
        if  W.flag
        then
                REFERENCE := W.value;
                COUNTER := 0;
                out  open_VALVE;
                goto 2
        else
                goto 1
        end

State 2
```

```
        if  MINUTE.flag
        then
                H.value := read H;
                if (REFERENCE>H.value)
                then
                        out   close_VALVE ;
                        out   DURATION
(COUNTER);
                        goto 3
                else
                        COUNTER :=
COUNTER+1;
                        goto 2
                end
        else
                goto 2
        end

State 3
        goto 3
```

Remark:

In this automaton, the management of the sensor H has been simplified and reduced to the reading of its value. In fact, when a sensor is needed for a reaction, its value must be read once and only once during the current instant. This constraint induces some extra code.

II.3 EXECUTION OF A REACTION

The compiler can translate an ESTEREL program into a sequential language which implements the automaton (structure, variables and the 'execution engine'). For the sake of simplicity, we consider that we have a unique automaton.

The hypothesis of instantaneous reaction could be satisfied by an infinitely fast machine. Of course, it is an ideal case. Practically, data processing is non-zero duration. They are executed sequentially, but the sequence corresponding to **a reaction must be indivisible**. The physical duration of an "instant", is the duration of the indivisible sequence associated.

Since reactions must not overlap, the executions of the automaton have to be controlled. An atomic reaction is triggered by an extra signal called "**activation** (of the automaton)".

So, to execute a reaction, one proceeds as follows:

1- Built up the current event
2- Activate the automaton
3- Execute **the** transition

The first step prepares an image of the environment. All the signals set during this phase are considered as simultaneous and they behave as if they occur at the very beginning of the current instant. The activation of the automaton starts the current instant. It is the responsibility of the execution machine to decide when the activation has to be done. Since the automaton is deterministic, given the current state and the image of the environment, a unique transition is possible. All the actions associated with this transition are executed, changing internal variables and calling output functions. When the sequence is terminated, the automaton enters the next state and all the input flags are reset. Then, the automaton returns the control to the caller; the current instant ends.

II.4 THE 'EXEC' STATEMENT

Sending the order to open the valve (statement **emit** open_VALVE) is a simple action consisting in setting a bit on some physical port. This action is sufficiently fast to be taken as instantaneous with respect to the physical process. Sometimes, actions need a non negligible amount of time, e.g complex computations [CosFav90] or communications [AndFan90].

Example:

In a flexible manufacturing system, let MOVE be a transportation operation. A part must be carried from a current place to another one. A status is returned on the completion of this action.

An incorrect solution is ESTEREL would be:

```
type  POSITION; % externally defined
procedure  MOVE (integer) (POSITION);
% externally defined
.....
var RES:integer, TO: POSITION in
        ......
        call MOVE (RES) (TO);
        ......
......
```

The execution of a procedure in ESTEREL is supposed to be instantaneous. At the automaton level, the execution of the procedure MOVE is done in an indivisible way, during the transition. Thus, actions following MOVE cannot be executed before the termination of MOVE. The requirement of negligible reaction time is obviously not met.

A solution could be to define an external (asynchronous) task T_MOVE. Such a solution has been presented in [AndFan90]. In the task T_MOVE, the body of MOVE is executed on the occurrence of a signal 'start'. When the body is done, the task emit a signal 'terminated' back to the automaton.

```
type POSITION;
......
input  terminated_T_MOVE (integer);
output  start_T_MOVE (POSITION);
......
var RES:integer, TO: POSITION in
        ....
        %call MOVE  (RES)  (TO);  replaced by
....
        emit  start_T_MOVE (TO);
        await  terminated_T_MOVE;
        RES:=?terminated_T_MOVE;
        ......
......
```

In this example, the expansion of the call procedure into a sequence 'emit; await' solves the problem of the duration, because 'emit' is instantaneous and does not prevent other actions from executing.

Now, suppose that one imposes an upper bound (Dmax) for the duration of MOVE. The execution of MOVE has to be guarded by a '**do…watching'** statement. If the timeout occurs, according to the semantics of this instruction, the body must instantly terminate. Therefore the asynchronous execution of T_MOVE must be aborted. A 'kill' signal must be generated. A special statement is available in the new release of ESTEREL: the **exec** statement [Paris91]. The previous example might suggest that **exec** is only a macroinstruction, easily expanded to classical ESTEREL statements. That is not generally true. The '**exec**' statement is actually a new primitive statement and the compiler takes it as such.

```
type  POSITION; % externally defined
procedure  MOVE (integer) (POSITION);
% externally defined
......
var RES:integer, TO: POSITION in
        ....
        do
                ......
                exec MOVE (RES) (TO);
                ......
        watching Dmax S
                % S occurs every second of time
        timeout  .......
        end
......
```

The compiler generates the 'start' and 'kill' signals automatically. If a sequence 'start;kill' has to be emitted at the same instant, no emission is done. Note that a sequence 'kill;start' is not discarded, because it is an immediate reinitialization of an external task.

III. ABSTRACT EXECUTION MACHINE

OVERVIEW

In this section, we make abstraction of the actual processor(s) on which the ESTEREL program may run. The abstract execution machine is composed of

1) an input processing unit;
2) an output processing unit;
3) a reactive machine;
4) an asynchronous machine.

Components 1 and 2 are not characteristic of ESTEREL, they provide interfacing facilities. Component 3 is the heart of the machine, it executes reactions. The fourth component is optional, it is used for possible exec statements.

On FIG.1, information transfers are denoted by links. Thin arrows stand for transfers occurring in one (ESTEREL) instant. Thick edges represent asynchronous transfers. In what follows, we characterize each functional unit and we suggest different control strategies for these units.

III.1 INPUT /OUTPUT PROCESSING

Reactive systems and real-time systems are event-driven. Most of the events are coming from the outside. The input processing unit extracts this information from the environment, collects it and makes it available for further processing. The output processing unit allows reactions of the controller on its environment.

The hardware interfaces are generally controlled by dedicated software. The most classical strategies for information collection are polling of inputs (program-controlled acquisition) and interruption handling (event-driven acquisition). Analogous solutions exist for information broadcast.

III.2 THE REACTIVE MACHINE

The input processing unit maintains a vision of the environment (information storage). This image is updated asynchronously, which is not convenient for the synchronous evolutions. The reactive machine must sample input signals from the storage unit, and then, execute the reaction according to the ESTEREL program. There exists some latitude in how the information is taken and when the automaton is activated.

Functionally, one may consider three units in the reactive machine: the event generator, the automaton and the synchronous action processing unit.

III.2.1 The Event generator

This component is the "history maker". It is in charge of preparing the input signals for the automaton (current input signals). When the current input signals are set, the activation of the automaton may be done. No further changes in the input signals are allowed for the current instant.

III.2.2 The automaton

The automaton is a machine able to memorize a present state, to compute a next state according to the current input signals and sensor values, and to control the execution of the actions. Both hardware and software can implement automaton efficiently.

III.2.3 The synchronous action processing unit

This unit executes the actions associated with the transition, under the control of the automaton. Recall that the ESTEREL compiler generates external references. These references are resolved by the synchronous action processing unit. Actions are said to be synchronous because they are fully executed during one ESTEREL instant. Hardware co-processors are surely adequate implementations to this processing.

III.2.4 Strategies

Of course, at each ESTEREL instant, the execution of the event generator must precede the reaction (automaton and actions). The reaction is fully deterministic because it is the exact behaviour imposed by the program. The choice of the input signals and the activation of the automaton, on the contrary, may be critical. The automaton reacts to the events submitted by the event generator. The event generator should make a history of the evolutions of the environment "as realistic as possible". An inadequate strategy for the event generator will badly modify the behaviour of the execution machine.

Two extreme policies are **self-synchronization** and **clock-driven** systems. In the first solution, as soon as the input processing unit updates an input signal S, the event generator is called, putting S as the unique current input signal. Then the automaton is activated. That seems to be very close to the ideal synchronous reaction. Unfortunately, there exists a chance of losing stimuli. We have to use a queue which presents events to the event generator in a first in, first out order. The second solution supposes an external "clock"* that activates the automaton at every tick. Of course, the amount of time between two successive ticks, must be longer than the duration of a reaction. In this case, all the events occurring in between two successive ticks are considered as simultaneous. Grouping signals may imply additional verification if some relationships between signals (exclusion, implication) have been declared in the program.

III.3 THE ASYNCHRONOUS MACHINE

This part is optional. It is intended to execute non instantaneous actions. If one wants to preserve the safety and the determinism brought by the synchronous language, only well identified actions should be put in this asynchronous unit.

III.3.1 The Exec manager

The 'Exec manager' receives signals emitted from the reactive machine (start and kill signals) and controls several tasks according to these requests. The exec manager is informed when a task terminates. It sends the results and a termination signal back to the information storage.

The manager must have functionalities of an executive kernel and especially multi-tasking facilities if the number of running tasks is greater than the number of processors.

III.3.2 The asynchronous actions

Recall that those actions are said to be asynchronous because they are not executed in an ESTEREL instant. The simplest actions are computations taking a fixed amount of time (e.g digital filtering). They may be considered as co-routines, and their behaviour is basically deterministic. Some actions may have non deterministic behaviour. They need special attention particularly if they may communicate to each other. In this case, the reactive machine acts as a scheduler for asynchronous tasks.

---

* Any repetitive event may be taken as a "clock", e.g an electronic oscillator, keystrokes on a keyboard, …

IV. EXAMPLES OF IMPLEMENTATIONS

IV.1 UNIX-SYSTEM

Some execution machines have been developed at CMA ([Berry2.89], [CosFav90], …). They are reactive but not submitted to critical time constraints.

There exist two kinds of such machines: simulation-oriented and application-oriented. The main difference is that the strategy, coded in the 'event generator', is fixed for the applications, whereas the user can modify the strategy for the simulations.

Applications can also be divided into: "simple applications" without asynchronous communication and "big applications" where the different parts of the application are in separate UNIX processes. When an application executes on a unique processor, we can distinguish three Unix processes:

the first deals with 'input processing',

the second is the task manager,

and the last gathers the 'event generator', the 'synchronous actions processing' and the 'automaton'.

Of course, a process must be associated with each asynchronous task and with the 'exec manager' [Paris91].

Now, considering a distributed implementation, the 'event generator', the 'synchronous action processing', and the 'exec manager' have to be split (and distributed). The different parts must communicate. The UNIX environment provides adequate facilities, viz. the Inter-Process Communication known as Remote Procedure Call (RPC). This communication can be considered as a distributed pipe. In fact, the execution of an application on one processor is a particular case of the distribution. RPC allows the distribution as well as the local execution. That is a great advantage in terms of coding. Moreover the time overhead due to the communications can be ignored in case of local execution.

IV.2 REAL-TIME CRAFT

Real-Time Craft (RTC) [GSI87] is a SCEPTRE-like [BNI84] real-time multitasking executive. We have used it for a mixed (synchronous / asynchronous) implementation of an intelligent controller. Our objective was to improve the performance and the predictability of the controller. The target machine was imposed (a microcomputer). The application was both reactive and transformational. The reactive part has been programmed in ESTEREL. The transformational one has been conceived as a collection of communicating tasks managed by RTC. For details refer to [AndFan90].

In this kind of application, all the components of the executive machine are implemented on the same processor. Multi-tasking facilities are necessary. Input and output processing is done by "immediate" tasks (interrupt handlers). The asynchronous machine consists of asynchronous tasks and the RTC executive. The reactive machine is a distinguished task, say Autom, which is waiting for events emitted from other tasks.

On the occurrence of an event (stimulus), Autom executes an indivisible sequence consisting of setting up input signals and performing the transition. The indivisibility is ensured by an exclusion mechanism (region in SCEPTRE) which inhibits the RTC scheduling in order to prevent Autom from pre-emption. The task Autom is given the highest priority, so that it can react promptly to stimuli. In fact, the task Autom, which implements the reactive machine, acts as a "reactive scheduler" with respect to the application. The scheduling of the asynchronous tasks is entrusted to the RTC executive, as usual.

V. CONCLUSION

Synchronous approaches simplify the design of reactive systems. Most of the real-time systems present reactive features. So, a language such as ESTEREL is enticing for the design of real-time systems. The main impediment is that the real world is basically asynchronous. It is a necessity, at the implementation level, to interface the asynchronous environment to the synchronous world of the automaton generated by the ESTEREL compiler.

In this paper, we have suggested the architecture of what could be an execution machine for ESTEREL, and other synchronous languages, as well. The machine has been defined functionally, in an abstract way. It is too early to assert that this abstract machine is a guide line to the future implementations. New implementations, on non classical architectures (RISK micro-controllers) and on distributed systems, are now in progress. Those experiences may alter our opinion on what the abstract machine must be.

Beyond implementations issues, general problems about correctness have come to light. Synchronous realizations can be proved to be correct when executed synchronously. The proof techniques used are not directly applicable to the asynchronous actions, and especially to the '**exec**' statement. Fundamental reflections have to be deepened in order to master relationships between synchrony and asynchrony.

VI. REFERENCES

[AndFan90] **C ANDRE, L FANCELLI**, *"A Mixed Implementation of a Real-time system"* , EUROMICRO'90, Amsterdam, August 1990.

[BeCoGo87] **G BERRY, P COURONNE, G GONTHIER**, *"Synchronous programming of reactive systems: an introduction to ESTEREL"* , Rapport de recherche INRIA n° 647, March 1987.

[Berry1-89] **G BERRY**, *"Real-time programming: special purpose or general purpose languages"* , in Information Processing 89, G.X Ritter (Ed.), Elsevier Science Publishers B.V (North Holland), 1989.

[Berry2-89] **G BERRY**, *"Programming a Digital Watch in Esterel v3, Esterel v3 Programming examples"* , Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1989.

[BNI84] **Bureau d'Orientation de la Normalisation en Informatique**, *"SCEPTRE: proposition de noyau normalisé pour les exécutifs temps-réel"* , TSI, vol 3, n° 1, 1984.

[Cisi90] **CISI Ingéniérie**, *"ESTEREL Reference Manual"* , January 1990.

[Dier90] **D DZIERZGOWSKI**, *"Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient"* , TSI, vol 9, n°4, pp289-312, 1990.

[CosFav90] **E COSTE-MANIERE, B FAVERJON**, *"A programming and simulation tool for robotics workcells"* , Intern. Conf. on Automation, Robotics and Computer Vision (ICARCV 90), Singapour, September 1990.

[GSI87] **GSI-Tecsi** *"RTC: XEC86 Executive user's guide"* , Mai 1987.

[HaPn85] **D HAREL, A PNUELI**, *"On the development of Reactive Systems, Logic and Models of Concurrent Systems"* , Springer-Verlag, pp. 477-498, 1985.

D HAREL, *"Statecharts: a visual formalism for complex systems"* , Science of Computer programming, 8(3), pp231-274, June 1987.

J.P PARIS, "*Communications Synchrones Asynchrones. Application à ESTEREL*" , Thesis, to appear in 1991.