Simulation of Temporal Behaviour

Charles ANDRE

June 26, 1993

Laboratoire Informatique, Signaux, Systèmes
I3S
Université de Nice-Sophia Antipolis / CNRS
41 bd Napoléon III
F 06041 Nice cédex, France
e-mail: andre@cma.cma.fr

Abstract

The amount of time elapsed between the occurrence of an interrupting event and its service (latency time) must be fully predictable in hard real time systems. Stochastic characterization of latency is not sufficient. Realistic simulation of the temporal behaviour of the system may bring better information. Since interruption mechanisms are not easily supported by classical Discrete Event simulation, we propose an original solution to the simulation of interrupt-driven systems with priority. This approach combines an Object-Oriented General Purpose Language and an imperative Synchronous Language.

Keywords: Discrete Events, Real-Time, Simulation, Synchronous language, Object-Oriented programming.

1 Introduction

For an ever-increasing number of computer-based systems, giving correct results is not sufficient. A late generation of information is often a faulty situation. Systems subject to temporal constraints are known as *Real-Time Systems* (RTS, for short). Though timeliness is not necessarily the first demand for a RTS, we shall restrict our attention to the verification of temporal constraint satisfaction. Safety, dependability and predictability are, at least, as important as temporal requirements [1], they are not addressed in this paper.

In fact, the phrase "real-time system" covers a wide range of systems so that issues and solutions may greatly vary according to the kind of application. When deadlines are inviolable, RTS are said *"Hard RTS"*. The temporal behaviour of these systems must be *predictable*. The *response-time* of the computer system is a factor that acts upon predictability. On the one hand, many RTSs, like classical signal processing applications, process incoming signals in a periodic way. The response-time of the system can be easily derived from the actual code. On the other hand, for some RTSs, evolutions are triggered by *sporadic events*. If these systems are *interrupt-driven*, their actual temporal behaviour can be very complex and highly dependent on the load of the system.

Systems that maintain a permanent interaction with their environment are called *Reactive Systems* [2]. When reactions are temporally constrained, systems are *real time reactive systems*: incoming signals (stimuli) must be processed at a rate imposed by the environment and the reaction in response to the stimuli must be "prompt enough". Of course, an implementation of a real-time reactive system must be proved *temporally correct*. So, a precise knowledge about the temporal distortion introduced by the actual implementation is required.

This paper copes with the issue of the temporal behaviour prediction for real-time reactive systems like (micro-)controllers. Those systems are mostly interrupt-driven. Moreover, the handling (supported by hardware and/or software) is generally based on *interruption mechanisms involving priority and preemption*. Sequential models are not satisfactory because input signals are many and *occur concurrently*. The degree of emergency associated with some signals makes matters worse. Models dealing with both priority and concurrency are not widely available and they are not easy to analyze (e.g. Deterministic Stochastic Petri Nets [3]). Statistical results are not sufficient for hard RTS. Deterministic performance evaluation could be obtained at the expense of heavy computation. An alternative to the analytic approach is the *simulation*. However, concurrency and reactivity are still a problem for classical languages or general modelling tools. So, we advocate a new approach to the simulation of *Discrete Event Systems* with concurrent and reactive behaviour. We propose a combination of an *Object-Oriented General Purpose Language* (C++[4]) and an *Imperative Synchronous Language* (ESTEREL[5]).

The paper is organized as follows. In the first part we characterize the kind of systems we are interested in. The second part describes the architecture of the simulator with its two components: the transformational and the reactive ones. Since the transformational part of the simulator is somewhat classical, we focus on the reactive part in the third section. Quantitative results and futher studies are presented in the conclusion.

2 Analysed Systems

2.1 Motivations

We started up with the issue of implementing efficient execution machines for ESTEREL programs. Ref [6] introduces "execution machines" in an abstract way. At the implementation stage, dependability and efficiency are major concerns. ESTEREL programs are compiled into automata (see [7]). Since the implementation of an automaton is a well-solved problem, the main cause of inefficiency may be a poorly-designed interface between a physical environment and the automaton. Because of real-time constraints, we were interested in the temporal behaviour of our interfaces. It appeared that this problem could be stated in a more general way, in terms of performance evaluation of interrupt-driven systems.

2.2 Interrupt-Driven Systems

We want to study the temporal behaviour of systems characterized by

1. numerous inputs,
2. sporadic events occurring concurrently,
3. interrupt-driven processing,
4. preemtive processing,
5. with fixed priority.

A generic model, similar to hardware architectures using PICs (Programmable Interrupt Controllers) can be proposed (Fig. 1). The inputs are structured as a two-layer hierarchy of ports (`PORTS`). Physical signals are presented on physical ports. Each port is characterized by a level and a sub-level implying its priority. An occurrence date, a type and a value are associated with each physical signal. Processings are carried out by (interrupt) handlers associated with levels. This model supports the two interrupt techniques commonly used: *vectoring* and *polling.*

Each port is denoted by a couple of integer values: <level.sublevel>. The number of levels (`LevelNb`) and sub-levels (`SublevelNb`) are parameters.

Definition 1 ***Priority ordering***
Let $PORTS = \{<l.s> \mid l \in [0..LevelNb-1], s \in [0..SublevelNb-1]\}$.
$\succ \subset PORTS \times PORTS$ *is defined as follows:*
Let $p_1, p_2 \in$ `PORTS`, *such that* $p_1 = <l_1.s_1>, p_2 = <l_2.s_2>$

$$p_1 \succ p_2 \Longleftrightarrow (l_1 < l_2) \vee ((l_1 = l_2) \wedge (s_1 < s_2)).$$

The highest priority port is the <0.0>. Signals received on port $<l.s>$ are processed by `HANDLER[l]`. The background task is assumed to be the lowest priority (`LevelNb`).

Figure 1: the Interface Model

The temporal behaviour of the handler, whose number is l, is characterized by four parameters:

- σ the context saving time,
- λ_l the duration before acknowledging the interrupt request,
- δ_l the duration for the complete handling,
- ρ the context restoration time.

Parameters σ and ρ are constant parameters. They correspond to non interruptable sections of code. λ and δ depend on the handling level. The values of these parameters suppose a full availability of the processor. In the presence of interruptions, the actual durations may be longer. For the port $<l.s>$, $\Lambda_{l,s}$ denotes the effective duration between the occurrence of the input signal and the acknowledgment to the interrupt request; $\Delta_{l,s}$ is the actual amount of time between the occurrence of the input signal and the completion of its handling. Fig. 2 shows a possible evolution in a simplified situation characterized by 4 levels and 0 sublevel. B stands for the background task. The contents of the system stack are represented below the timing diagrams. Cx is the context of handler x.

Definition 2 ***Interrupt Handling***
Let L be the level of the currently running handler (possibly `LevelNb`*).*

Figure 2: Temporal Behaviour

- *Whenever an event occurs on port* $<l.s>$ *do*
  `if` $l < L$
  `then` *preempt* `HANDLER[L]` *and execute* `HANDLER[l]`
- *When* `HANDLER[L]` *terminates, execute the highest priority pending request*

At a first look, an else-clause seems to be missing. In our systems, interrupt requests are held while not acknowledged. This can be easily done by hardware.

Note that, there is an interrupt masking facility. The interrupt handling can be suspended by the signal "`disable`" and resumed by "`enable`" generated by the background task, when executing. In fact, context savings and context restorations are enclosed within implicit `disable` and `enable`.

2.3 Expected Results

Our interfaces are characterized by concurrent and preemptive behaviours. There is no assumption about input signals, but the existence of a minimal interval of time between two successive occurrences of the same signal. Let $G_{l,s}$ be this interval for port $<l.s>$. Generally, inputs are not even independent. In such circumstances, *analytical methods* are helpless. Thus, we resort to *simulation.*

Another problem is raised by the fact that value of signals may be of different types and that the handling may depends on the history of signal values. Even powerful tools, like QNAP2, have difficulty in dealing with our two-layer hierarchy, and interrupt handling policy. That is the reason why we have developed a new approach to simulation, an approach that takes account of the reactive behaviour of the system to be analysed.

In the context of real-time systems, several temporal parameters are of interest. An implementation is said to work "in real-time" if it is fast enough to meet its temporal requirements. *Non violation of a deadline* is a usual requirement, especially stringent for hard-real time systems. In order to prove the absence of violation, we have to determine the *worst-case response-time*. This value cannot be given by a simulation. Fortunately, it can be expressed by a formula:

The worst-case delay between the occurrence of a signal on port $<l.s>$ and the completion of its processing in the `HANDLER[l]` is $\Delta^{wc}_{l,s} =$

$$\frac{\delta_{l,s} + \max\{\delta_{l,j} \mid s < j < SublevelNb\} - 1 + 2 * (\sigma + \rho)}{1 - \sum_{i=0}^{l-1} \sum_{j=0}^{SublevelNb-1} \frac{\delta_{i,j}}{G_{i,j}} - \sum_{j=0}^{s-1} \frac{\delta_{l,j}}{G_{l,j}} - (l * SublevelNb + s) * (\sigma + \rho)}$$

This formula assumes that $\forall <i.j> \succ <l.s>$: $\Delta^{wc}_{i,j} < G_{i,j}$. This value can be a very pessimistic one. The lower the priority, the more unlikely this value is. A more realistic evaluation of the response-time is desirable. Besides statistics about $\Delta_{l,s}$, other parameters can be obtained by simulation:

- statistics about $\Lambda_{l,s}$,
- existence of "*overrun*", i.e. a new signal arrives on port $<l.s>$ before the previous one has been acknowledged $(\Lambda_{l,s} > G_{l,s})$,
- ...

When an overrun occurs, the old value is over-written. In this instance, the controller loses inputs, that can be tolerated for some ports but may be forbidden for others. By simulation, critical parameters can be found out. If the dynamics of the system to be controlled is given ($G_{l,s}$ given for all $<l.s>$), the designer may alter priority, recode handlers, or even change hardware in oder to tune its controller.

3 Simulator Overview

3.1 Components of the Simulator

The simulator is composed of two main parts:

1. the Input Signal Generation,
2. the Input Signal Handling

The first part is mostly *transformational*, whereas the second is essentially *reactive*. General Purpose Languages (GPL) are suitable for programming transformational programs but they lack reactive statements for reactive programming [8]. The converse is true for the synchronous language Esterel. So, our solution is a combination of both GPL and Synchronous Language approaches. An Object-Oriented GPL (C++) is used for part 1, Esterel expresses part 2. Fig. 3 shows an overview of the simulator structure.

Figure 3: Simulator Structure

3.2 The Transformational Part

We have defined parametrized *Signal Generators*. A data type is attached with each signal generator. A generator yields signals by the means of associated functions that provide occurrence date and value of signals. We have chosen C++ because of its object-oriented facilities. Our generators can support any kind of data types. The class "`Signal_Generator`" has two important member functions (Methods) called "`start`" and "`update`". The former allows to initialize a sequence of signal occurrences with a given periodicity, the latter triggers successive occurrences. Aperiodical signals are assigned an "infinite" period. Moreover the period is a function, not necessarily a constant, so that even non strictly periodic events can be generated, as well. Since this part is somewhat classical, we will not give further details.

3.3 The Reactive Part

This part deals with the "reactions". Esterel is dedicated to this kind of applications. It is a high-level language including classical control statements and supporting concurrency, exception and interruption handling. Esterel is defined by a mathematical semantics that determines a unique reaction to each stimulus, so that the composition of Esterel modules is fully *deterministic*, which is a valuable quality for real-time system programming. Esterel relies on the *Perfect Synchrony Hypothesis*, i.e. stimuli and reactions are assumed to be simultaneous, and information is instantaneously broadcast.

3.4 Time in Simulation

The *time base* is discrete and a date is denoted by a long integer. Some physical unit of time is understood. The simulator is a *dicrete event system simulator* partly *event-driven*, partly *clock- driven*. Events are occurrences of physical signals. A signal is produced by a signal generator which assigns an occurrence date to each physical signal occurrence. Events are stored in an ordered list "`schedule`". A distinguished signal generator is called "`theClock`". Each occurrence generated by `theClock` is interpreted as a tick of the "*system clock*". The system clock is significant for the reactive part and is used for a clock-driven simulation of this part. In fact, the system clock is a subset of the time base. For instance, the unit for the time base may be 10 ns, whereas the system clock is given a period equal to 200 ns. The periodicity of the system clock can be easily modified, even in a dynamical way when some slow or speed-up motion is wanted during the simulation.

3.5 Synchronisation kernel

The heart of the simulator is an infinite loop. First, all the events that have occurred since the previous system clock tick, are collected. They are seen as simultaneous by the system. Then, a reaction is executed (evolution of the reactive part) according to the collected events. The concurrent executions of the signal generators and the handlers are simulated by a event-driven mechanism. Generated events are put into the ordered list "`schedule`". The insertion key is the occurrence date. When an event is scheduled at a date already used, this event is inserted *before* the other event(s) with the same time stamp. Occurrences of `theClock` are enqueued in the same list, but in case of equality of date, the occurrence of the system clock tick is scheduled *after* the other event(s) with the same date. Events are extracted from the list according to a "*smallest date first*" discipline. This enqueing/dequeueing strategy implies that when `theClock` is dequeued from the list `schedule` there is not any event in the list still to be scheduled at this instant. Whenever a signal is extracted from the list, the "`update`" method of the associated signal generator is invoked. This method generates a new signal occurrence which is enqueued in `schedule` for further use. The following program sums up the way the simulator runs:

```
do {
  while ((s=schedule.dequeue()) != theClock) {
      // s is a reference to an event
      signal the occurrence to the associated port
      update the associated signal generator
   } // here s== theClock: it's a new instant
   execute a reaction of the reactive part
   update theClock
} forever
```

The user has to define its configuration. Some parameters have to be statically defined (number of levels and sub-levels). If non predefined types are used their specification and their implementation in C++ must be provided. In order to facilitate this generation, a generic type is available. Signal generators can be instanciated interactively, while the simulation is running. Results are kept in files. For each physical signal occurrence, we have its value, its occurrence date, its possible over-writing date (if overrun), its acknowledge date, its delivery date (complete processing done). Since our simulator is only an experimental one, facilities for statistical analysis are not provided.

4 Programming of the Reactive Behaviour

4.1 Esterel Modules

The reactive part is decomposed into ESTEREL modules (Fig. 4):

- The `GATE` module which is in charge of collecting physical signals and issueing acknoledgement or overrun error messages.
- The `HANDLER` module which processes the incoming signals and generates logical signals.
- The `MULTIPLEXER` module. It is the heart of the reactive part. It manages the connection between instances of the GATE and the HANDLER modules.
- The `BACKGROUND` which simulates the temporal behaviour of the process that consumes processed signals.

4.2 Structuration of the Reactive part

There is `NB_LEVEL*NB_SUBLEVEL` instances of the module `GATE`, `NB_LEVEL` instances of the module `HANDLER`, one instance of `MULTIPLEXER` and one of `BACKGROUND`. Those modules are cooperating, they interact by (ESTEREL) signals. Contrary to classical approaches based on task cooperation (high-level language like ADA, or use of a sequential language, like C, calling Real Time Executive services), in Esterel, synchronization and communication are transparent to the programmer. The programmer can concentrate on the logics of the program, instead of being disturbed by synchronization and communication mechanism invocations.

Figure 4: Internal Structure of the Simulator

The resulting Esterel program is composed of lots of parallel modules. The explosion of the automaton generated by the compiler is inevitable. Classical solutions to avoid explosion of Esterel programs are the *cascade mode* and the *array mode*. The *array mode* allows to produce a small automaton instead of a big one resulting from the product of independent automata. Of course, the many instances of the same module must be independent in order to apply the array mode. So are the Gate and Handler modules. The *cascade mode* is useful when there exists a partial ordering on the signal exchanged between modules. In this case it is possible to generate several small automata that are executed in a fixed ordering. Unfortunately, few Esterel programs are liable to be compiled in this mode, because the compiler determines a possible linear ordering statically, by syntactic analyses only. Our simulator needs a combine use of the cascade and array modes, but these two options are exclusive.

To overcome this drawback, we suggest a *partially asynchronous* implementation. The simulator is implemented as several automata which are executed, at each logical instant (system clock tick), in a fixed order. The signals are exchanged asynchronously between automata. Note that the asynchrony is restricted: signals are, at most, delayed by one instant. This mode is close to the one underlying CSML[9].

4.3 Effective programming of the Reactive part

The simulator is composed of cooperating parts:

- Automata, themselves decomposed into
  - `GATE_autom` generated by the compilation of the GATE module in the array mode,
  - `HANDLER_autom` generated by the compilation of the HANDLER module in the array mode,
  - `MULTIPLEXER_autom` generated by the compilation of the MULTIPLEXER module,
  - `BACKGROUND_autom` generated by the compilation of the BACKGROUND module.
- Data Structures: types implementations, ordered lists for messages, output signal functions, ...

The different automata are executed, during the same logical instant, in a fixed order:

1. All the `GATE_autom`
2. The `MULTIPLEXER_autom`
3. All the `HANDLER_autom`
4. The `BACKGROUND_autom`

Fig. 4 illustrates the different components and their interconnections. The signals are explained in Section.4.4. The communication between automata is realized by calls to input function of an automaton within one output function of another one. For instance, `GATE_autom` sends a signal `UPDATED` to `MULTIPLEXER_autom` by executing its output function `GATE_O_UPDATED()`:

```
GATE_O_UPDATED(n)
int n;
 { MULTIPLEXER_I_UPDATED(); }
```

4.4 Esterel modules

4.4.1 The Gate module

```
%       Charles ANDRE
%       Jan 20, 1993
%       V6
%
```

Figure 5: The GATE module

```
% basename GATE
% use the -array option
% input by GATE_I_IT(gate_number)
%
%  a reference to the message is entered the waiting queue
%  while the I_ function is called
%
module GATE:
%
% declarative part
input   IT, D_IT_ACK;
output  ACK, OVERRUN, UPDATED;
%
% imperative part
  loop
    await immediate IT;
    emit UPDATED;
    await
       case D_IT_ACK do
            emit ACK
       case IT do
            emit OVERRUN
    end await
  end loop
end module
```

4.4.2 The Multiplexer module

```
%       Charles ANDRE
%       Jan 20, 1993
%       V6
%
% MULTIPLEXER concentrates information from  GATEs
% and dispatch to HANDLERs and BACKGROUND
%
module MULTIPLEXER:
% declarative part of  MULTIPLEXER
  type Date;
  function smallest_int (integer, integer): integer;
```

Figure 6: The MULTIPLEXER module

```
  input H, ENABLE, DISABLE, UPDATED, NONINTERRUPTABLE,
        D_ACTIVE (combine integer with smallest_int);
  output  SH:integer;
  constant NB_LEVEL: integer;
  sensor HIGHEST: integer;
  %
  % ?HIGHEST is the highest priority channel waiting for
handling
  %  (i.e smallest number)
%
% imperative part of  MULTIPLEXER
  signal TOP_LEVEL:integer, H_FILTERED in
    every UPDATED do
      emit TOP_LEVEL(?HIGHEST)
    end every
    ||
    every H do
      present NONINTERRUPTABLE else
          emit H_FILTERED
      end present
    end every
    ||
    loop
     do
       every H_FILTERED do
         emit SH(NB_LEVEL) % no interrupt service
       end every
     watching ENABLE;
     await immediate H_FILTERED;
     do
       loop
        emit SH( smallest(?TOP_LEVEL, ?D_ACTIVE))
       each H_FILTERED
```

```
      watching DISABLE
    end loop
  end signal
end module
```

4.4.3 The Handler module

Figure 7: The HANDLER module

```
%       Charles ANDRE
%       Jan 20, 1993
%       V6
%
% version with self-recognition
% basename HAND
% use the -array option
%
% implicit suspension by SH control
%
module HANDLER:
% declarative part of  HANDLER
  input H, SH;
  output ACTIVE, UPDATED, NONINTERRUPTABLE;
  output IT_ACK: integer;
  %
  constant SIGMA: integer, RHO: integer;
  procedure pop (integer) ();
  procedure esterel_sig_gen()(integer);
  function latency (integer) : integer;
  function duration (integer) : integer;
  %
  relation SH => H;
  %
  % H general clock conveying the current date
  % SH selective clock: present when to be executed
  % ACTIVE emitted when running and not terminating
  % UPDATED emitted when starting the IT service
  %
  % latency returns the nb of clock cycles before IT_ACK
  % duration returns the nb of clock cycles for full execution
```

```
    % arguments of esterel_sig_gen, latency, duration identifies
    %  the interrupt handler (its priority level)
    %
    % esterel_sig_gen:
    %     in: channel number,
    %     generates the true ESTEREL signals
    %     out: void
    % pop : in: void,
    %       out: channel number
  %
  % imperative part of  HANDLER
    var k:integer in    % k is the channel number
      loop
        await SH;
        call pop(k)();
        emit UPDATED;
        do
          do
             sustain NONINTERRUPTABLE
          watching SIGMA H; % context saving
          await latency(k) SH; emit IT_ACK(k)
           ||
          loop emit ACTIVE each H
        watching duration(k) SH;
        call esterel_sig_gen () (k);
        do
           sustain NONINTERRUPTABLE
        watching  RHO H
      end loop
    end var
  end module % HANDLER
```

4.4.4 The BackGround module

Figure 8: The BACK GROUND module

```
%       Charles ANDRE
%       Jan 20, 1993
```

```
%       V6
%
%
% basename BKGND
%
% special HANDLER with lowest priority: NB_LEVEL
% implicit suspension by SH control
%
module BACKGROUND:
% declarative part
  input H, SH;
  output ENABLE, DISABLE, ACTIVE;
  constant NB_LEVEL: integer;
  %
  % H general clock conveying the current date
  % SH selective clock: present when to be executed
  % ACTIVE emitted when running and not terminating
  % always present for XECMACHINE
  % ENABLE allows interruptions, DISABLE forbids
%
% imperative part of  XECMACHINE
%
  every H do
     emit ACTIVE
  end every
||
   emit DISABLE;
   await 3 SH; % an arbitrary transient state, just to
see
   emit ENABLE
% free running
%
end module % BACKGROUND
```

References

[1] W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems.* Amsterdam: Kluwer Academic Publishers, 1991.

[2] D. Harel and A. Pnueli, "On the development of reactive systems in logic and models of concurrent systems," *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, vol. 13, pp. 477–498, 1985.

[3] M. Ajmone Marsan, G. Chiola, and A. Fumagalli, "Improving the efficiency of the analysis of DSPN models," in *Advances in Petri Nets 89*, (G. Rozenberg, ed.), pp. 30–50, Springer-Verlag, 1990.

[4] B. Stroustrup, *The* C++ *Programming Language.* Addison Wesley Publishing Company, 1986.

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

[6] C. André, J.-P. Marmorat, and J.-P. Paris, "An execution machine for ESTEREL," in *ECC'91*, (Grenoble, France), pp. 1672–1677, Hermès, July 1991.

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

[8] G. Berry, "Real-time programming: Special purpose or general purpose languages," (San Francisco), pp. 11–17, 11^{th} IFIP World Congress, 1989.

[9] E. Clarke, D. Long, and K. McMillan, "A language for compositional specification and verification of finite state hardware controllers," *Proceeding of the IEEE*, vol. 79, pp. 1283–1292, September 1991.