Programming Languages For Hard Real-Time Embedded Systems

J. Forget, F. Boniol, D Lesens, C Pagetti, M. Pouzet

Programming Languages For Hard Real-Time Embedded Systems

J. Forget[1], F. Boniol[2], D. Lesens[3], C. Pagetti[1], M. Pouzet[4]

1: ONERA/CERT, 2, avenue Edouard Belin, BP 74025 - 31055 Toulouse cedex 4 - France
2: ENSEEIHT, 2, rue Charles Camichel, BP 7122 - 31071 Toulouse Cedex 7 - France
3: EADS Astrium Space Transportation, Route de Verneuil - BP 3002 - F-78133 Les Mureaux Cedex - France
4: LRI, Université Paris-Sud 11, Bat. 490, 91405 Orsay Cedex - France

Abstract: Hard real-time embedded systems have traditionally been implemented using low level programming languages (such as ADA or C) at a level very close to the underlying operating system. However, for several years now the industry has started using higher level modelling languages, at least for early simulation and verification steps. The objective of this paper is to study existing formal languages including high level real-time primitives. Our review is built on the case study of an aerospace automated transfer vehicle, the particularity of which is to be composed of several multi-periodic communicating processes. In this paper, we emphasize the strengths and weaknesses of existing programming approaches when implementing this kind of system. As a result, the choice of the base rate of the program appears to have a major influence, not only on the difficulty to program the system correctly but also on the execution platform required to execute the program (operating system, scheduler, ...).

Keywords: Embedded, Real-Time, Programming Language

1. Introduction

An embedded system is a computer system embedded in a complete system including hardware and mechanical parts. Its purpose is to handle the physical system in its environment. Such systems are dedicated to some specialized functionalities on the contrary to general purpose computers like personal computers. Nowadays, embedded systems range from heavy industries like aeronautics, aerospace or railways, to lighter industries like home appliances or cell phones.

In the following, we are mainly interested in *reactive systems.* A reactive system interacts with its environment by repeating indefinitely the same sequence: acquiring data on sensors, processing data, producing data to control actuators. Such systems must respect *hard real-time constraints*, meaning that failing to respect these constraints can lead to catastrophic situations. These systems are required to be *functionally deterministic,* meaning that they always produce the same output sequence with respect to the same input sequence. They must be *temporally deterministic* as well, always having the same predictable temporal behaviour. While respecting these strong constraints, the system still needs to be optimized, in terms of latency, hardware cost, power consumption or weight for instance. Currently, such systems are mainly prototyped with high-level approaches (Matlab/Simulink [17] or the Synchronous approach [1]) and then programmed separately with low level languages, like C or Ada, at a level very close to the underlying Operating System. However, the complexity of the development process, due to the constraints mentioned above, calls for higher-level, formal programming languages, which cover the complete process from design to implementation.

We review such formal languages by studying the programming of an automated aerospace transport vehicle. The distinctive feature of this case study is its multi-rate aspect. Programming a reactive system using high level languages consists in describing the computations performed during a basic iteration, which is repeated indefinitely. When dealing with multi-rate systems, the choice of the rate of this basic iteration (chosen among the different rates of the system) leads to very different styles of programming, regarding real-time constraints but also communications between processes of different rates.

Synchronous languages are based on a paradigm well adapted for programming critical reactive systems but do not provide natively primitives for expressing hard real-time constraints. In practice, the multi-rate aspects have to be “manually” handled. The usual solution is to program using the fastest rate of the system as the basic iteration rate. However, this requires manual scheduling and splitting of slow processes into several fast sub-processes (“manual preemption”), which is tedious and error-prone. On the opposite, we cannot program the system correctly choosing the slowest rate as the basic iteration, due to the lack of primitives to control the scheduling of fast operations that have to be repeated periodically inside the basic iteration.

Real-time extensions to the synchronous approach have recently been introduced following two directions. First, real-time aspects can be introduced in the language and handled by the compiler as such, by specifying processes execution times [10],

latency constraints and periodicity constraints [8], [11]. These extensions clearly fit our case study better, though they still need some improvements, in particular concerning multi-rate communications. Second, the synchronous approach usually does not rely on underlying Operating Systems much (the synchronisations and scheduling are directly handled by the synchronous compiler). However refining the synchronous execution model, by allowing preemption for instance [16], enables the use of existing scheduling tools, which facilitates the compilation of synchronous multi-rate systems.
The Time-Triggered Approach [9] can also be used to specify our case study, as proposed in Giotto [13]. Interestingly, Giotto favours programming using the slowest rate as the base rate, on the opposite to the synchronous approach. However, using this approach at such a high design level does not seem to be appropriate. It mainly lacks control on the execution order of operations and on communication processes. TTA seems to fit better at lower level, for instance as an execution model for the synchronous approach.
Our paper is structured as follows. We first present our case study, an automated transfer vehicle in section 2. We then study its programming using available languages. We pay special attention to synchronous languages in section 3 and study their recent real-time extensions in section 4. We present the time-triggered approach in section 5 and conclude in section 6.

2. Case Study : an Automated Transfer Vehicle

Our case study is an Automated Transfer Vehicle (ATV) designed by EADS Astrium Space Transportation for resupplying the international space station. We present an adapted version of the safety unit of the vehicle (not the real version). Its purpose is to supervise the main computing unit of the vehicle in order to detect possible failures. If a failure occurs, the safety unit stops the current operation and moves the vehicle to a safe orbit, waiting for further instructions coming from ground control. The safety unit is represented in Figure 1.

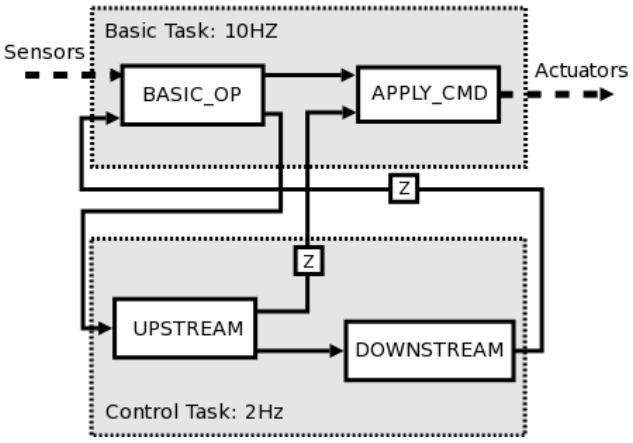

Figure 1: The case study

The safety unit is made up of two tasks, a fast task (`Basic Task`) executed with a rate of 10Hz, and a slow task (`Control Task`) executed with a rate of 2Hz. Each task is made up of two operations. The fast task acquires acceleration measures from sensors through the `basic_op` operation and sends them to the slow task. The `upstream` operation integrates successive acceleration values and computes the new required acceleration values, which are in turn sent back to the fast task. The operation `apply_cmd` computes new acceleration orders and sends them to the thrusters. The `downstream` operation adjusts acceleration values as the frame of reference (the vehicle) is moving and gives its feed back to `basic_op`. This whole process is repeated indefinitely.
The edges between operations model data dependencies. A data dependency implies a precedence relation between the communicating operations. The small “z” boxes represent delay communications. A delay communication means that the data produced during one iteration of the system is only consumed at the next iteration. Communications from the slow task to the fast task are performed using a delay to avoid the fast task to be delayed until the slow task is complete.
The programming of the processes performed inside the four operations of the safety unit is not the concern of this paper, we are mainly interested in dealing with the real-time and communication aspects of the system.

3. The Synchronous Approach

In the synchronous model, time is modelled as *logical time* separated into *instants.* The duration of an instant, as well as its start date, are not considered and remain abstract. Synchronous languages describe computations made during each instant (which are repeated indefinitely). The synchronous hypothesis states that computations made during one instant must end before the

beginning of the next instant. Among synchronous languages we can cite Esterel [2], Lustre/Scade [7], Signal [4], Lucid Synchrone[5] and Synchronous Data Flow [6]. In this paper, we focus on equational synchronous languages (Lustre/Scade, Signal, Lucid Synchrone).

3.1 Presentation

In equational synchronous languages, variables and expressions are *flows*. A flow is defined by its sequence of values, potentially infinite, and by its *clock*. The clock of a flow defines the instants when the flow is *present* (defined). Each time the flow is present, it takes the next value in its sequence of values. Flows are defined by equations and equations are structured into *nodes,* among which one is the *main node* (the entry point of the program). Nodes are organized hierarchically, they can be used in expressions to define flows.

Classic arithmetical and logical operators are extended point wise over flows. For instance, adding two flows produces a flow the value of which is the sum of the two flows at each instant. In the same way, `if-then-else` is the point wise conditional operator. The `pre` operator is an instant delay operator, it stands for the previous value of a flow. It is often used along with the `->` initialisation operator, as the first value of `pre x` is undefined. The `when` operator undersamples a flow using a Boolean condition. The flow `x when c` is present and takes the value of `x` only when `c` is true. Conversely, `current` replaces the absent values created by `when` by the last present value of the flow. This operator has been replaced by `merge` in Lucid Synchrone, which combines flows of complementary clocks. For instance, `merge c (x when c) (y when not c)` produces a flow the clock of which is the clock of `c`, and which takes the value of `x` when `c` is true and the value of `y` when `c` is false. The behaviour of these operators is illustrated Figure 2. Usually, lower level processes of the program are defined using imported nodes. Signatures of these nodes (input and outputs) are declared in the Lustre program, but the nodes are implemented outside, for instance using C code. Lustre only requires these nodes to have no hidden side-effect on the program: each imported node can only modify the values of the Lustre variables that are declared as its outputs (it can of course also modify internal variables not declared in the Lustre program).

`x`	x_1	x_2	x_3	x_4	…
`y`	y_1	y_2	y_3	y_4	…
`x+y`	x_1+y_1	x_2+y_2	x_3+y_3	x_4+y_4	…
`x->pre y`	x_1	y_1	y_2	y_3	…
`h`	True	False	False	True	…
`x when h`	x_1			x_4	…
`y when not h`		y_2	y_3		…
`current x`	x_1	x_1	x_1	x_4	…
`merge h x y`	x_1	y_2	y_3	x_4	…

Figure 2: Operators on flows. We give the value of each flow at each instant.

3.2 Application to the case study

In this section we program our case study using Lustre. We chose Lustre as it is the fundamental basis of Scade, often used for programming aeronautic and aerospace systems. The program (and encountered problems) would be fairly similar with other equational synchronous languages.

The main characteristic of our case study is that processes are performed at different rates. Consequently, we first have to choose the rate at which the basic iterations of the synchronous program will be performed: either the fast rate (10Hz) or the slow rate (2Hz). We use some auxiliary nodes, whose code is given Figure 3. Node `countN` is a counter modulo *n*. `everyN(n)` is *true* once each *n* activation. `never(c)` remains *false* as long as `c` has never been *true*, it remains *true* afterwards. Node `i_current` is an initialised delay: `i_current(c, init, f when c)` has value `init` as long as clock `c` has never been active; then it has the current value of `f` when `c` is active and the last active value of `f` otherwise.

```
node countN(n: int) returns(o: int)
let
  o=(0->(pre(o)+1)) mod n;
tel

node everyN(n: int) returns (reached: bool)
let
  reached=(countN(n)=n-1);
tel

node never(c: bool) returns (not_ever: bool)
let
  not_ever=(not c)-> (not c and pre(not_ever))
tel

node i_current(c: bool; init: int; x: int when
c) returns (o: int)
let
  o = if never(c) then init
      else current(x);
tel
```

Figure 3: Auxilliary Lustre nodes used to program the case study.

Programming with a fast base rate: We start with a version on the fast rate, the corresponding program is given Figure 4. This program assumes that the main node `msu_fast` is activated with a frequency of 10Hz (see section 3.3 for details). The nodes `basicOp` and `applyCmd` corresponding to the operations of the fast task are performed at each instant, thus at 10Hz. The nodes `upStream` and `downStream` corresponding to the operations of the slow task are performed once every five instants, thus at 2Hz, as they are activated only when the

clock `clock5` is true. Data communications from the fast task to the slow task cannot be consumed immediately at the instants when the slow task is not active; data produced is consequently memorised until the next activation of the slow task (flows `bop02`, `bop12`, `bop22`, `bop32`). Data communications from the slow task to the fast task (flows `ds` and `us1`) are delayed using the `pre` operator. The corresponding flows are set on the fast clock using the `current` operator.

```
node msu_fast(fromEnv, fromOtherMSU: int)
     returns (toEnv, toOtherMSU: int)
var clock5: bool;
    bop1,  bop02, bop12, bop22, bop32, bop42,
cur_ds, cur_us1: int;
    ds, us1, us2: int when clock5;
let
  bop32=0->pre(bop42);
  bop22=0->pre(bop32);
  bop12=0->pre(bop22);
  bop02=0->pre(bop12);
  bop1, bop42, toOtherMSU=basicOp(fromEnv,
fromOtherMSU, cur_ds);
  toEnv=applyCmd(cur_us1, bop1);
  clock5=everyN(5);
  us1, us2=upStream(bop02 when clock5,
                    bop12 when clock5,
                    bop22 when clock5,
                    bop32 when clock5,
                    bop42 when clock5);
  ds=downStream(us2);
  cur_ds=i_current(clock5, 0, pre(ds));
  cur_us1=i_current(clock5, 0, pre(us1));
tel
```

Figure 4: Programming the case study in Lustre, with a fast base rate (10Hz)

The main default of this version is that it assumes that it is possible to execute the four operations in less than the duration of an instant (100ms). Indeed, at cycles when both the slow and fast operations are activated, they must all finish before the beginning of the next instant.
This assumption is far too restrictive and leads us to a new version given Figure 5. This time we split the processes performed by the slow operations into several nodes (`upStream` split into `upStream0`, `upStream1`, `upStream2`, and `downStream` split into `downStream0`, `downStream1`) and distribute these nodes between five successive instants, which correspond to one slow cycle. Each of these nodes is activated only once every five instants, with an offset of one instant between the different nodes. Consequently, the complete slow task is indeed executed with a frequency of 2Hz.

```
node msu_fast2(fromEnv: int; fromOtherMSU: int)
      returns (toEnv, toOtherMSU: int; )
var clock0, clock1, clock2, clock3, clock4:
bool;
    count: int;
    bop1, bop02, bop12, bop22, bop32, bop42:
int;
    us_0: int when clock0;
    us_1: int when clock1;
    us1, us2: int when clock2;
    ds_0: int when clock3;
    ds: int when clock4;
let
  count=countN(5);
  clock0=(count=0);
  clock1=(count=1);
  clock2=(count=2);
  clock3=(count=3);
  clock4=(count=4);
  bop32=0->pre(bop42);
  bop22=0->pre(bop32);
  bop12=0->pre(bop22);
  bop02=0->pre(bop12);
  us_0=upStream0(bop02 when clock0,
                 bop12 when clock0,
                 bop22 when clock0,
                 bop32 when clock0,
                 bop42 when clock0);
  us_1=upStream1(current(us_0) when clock1);
  us1, us2=upStream2(current(us_1) when clock2);
  ds_0=downStream0(current(us2) when clock3);
  ds=downStream1(current(ds_0) when clock4);
  bop1, bop42,toOtherMSU=basicOp(fromEnv,
fromOtherMSU, i_current(clock4, 0, pre(ds)));
  toEnv=applyCmd(i_current(clock2, 0, pre(us1)),
bop1);
tel
```

Figure 5: Programming the case study in Lustre, with a fast base rate, spliting operations of the slow task

Our case study is a simplified system, however for a complete system, manually distributing processes of the slow task between five (fast) instants can be tedious and error-prone. This indeed requires the programmer to determine a fair distribution between instants in terms of execution times, which can be a difficult scheduling problem. This also assumes that the software architecture of the processes enable such a distribution. For instance, if a slow operation is split into two sub-nodes, as Lustre forbids side effects data produced by the first sub-node at the end of the first instant explicitly needs to be passed on to the next sub-node of the operation at the next instant (flow `us_0` between `upStream0` and `upStream1`). Such data communications can be numerous if the processes of one operation are very interdependent and this leads to high memory consumption.

Programming with a slow base rate: We will now consider a third version using the slow rate as the base rate of the program. The corresponding code is given Figure 6. This program assumes that the main node `msu_slow` is activated with a frequency of 2Hz. The nodes of the slow task (`upStream` and `downStream`) are executed at each instant. The nodes of the fast task are repeated five times for each instant in order to be executed with a frequency of 10Hz. The inputs and outputs of the program are respectively consumed and produced by the fast task, thus they are also duplicated five times. Data communications from the slow task to the fast task (`ds` and `us1`) are performed with a `pre` (delay).

```
node msu_slow(fromEnv0, fromEnv1, fromEnv2,
fromEnv3, fromEnv4: int;
              fromOtherMSU0, fromOtherMSU1,
fromOtherMSU2,
              fromOtherMSU3, fromOtherMSU4: int)
     returns (toEnv0, toEnv1, toEnv2, toEnv3,
toEnv4: int;
```

```
              toOtherMSU0, toOtherMSU1,
toOtherMSU2, toOtherMSU3,
              toOtherMSU4: int)
var us1, us2, ds: int;
    bop01, bop02,  bop11, bop12, bop21, bop22,
bop31, bop32,
    bop41, bop42: int;
let
  us1, us2=upStream(bop02, bop12, bop32, bop42);
  ds=downStream(us2);

  bop01, bop02, toOtherMSU0=basicOp(fromEnv0,
fromOtherMSU0, 0->pre(ds));
  bop11, bop12, toOtherMSU1=basicOp(fromEnv1,
fromOtherMSU1, 0->pre(ds));
  bop21, bop22, toOtherMSU2=basicOp(fromEnv2,
fromOtherMSU2, 0->pre(ds));
  bop31, bop32, toOtherMSU3=basicOp(fromEnv3,
fromOtherMSU3, 0->pre(ds));
  bop41, bop42, toOtherMSU4=basicOp(fromEnv4,
fromOtherMSU4, 0->pre(ds));

  toEnv0=applyCmd(0->pre(us1), bop01);
  toEnv1=applyCmd(0->pre(us1), bop11);
  toEnv2=applyCmd(0->pre(us1), bop21);
  toEnv3=applyCmd(0->pre(us1), bop31);
  toEnv4=applyCmd(0->pre(us1), bop41);
tel
```

Figure 6: Programming the case study in Lustre, with a slow base rate

This version is not correct because even if the fast operations are performed five times each slow cycle, nothing forces the different repetitions to execute at the right time during one instant. Le t_0 be the start date of the instant. The duration of the instant is 500ms (2Hz). We expect the fast nodes `applyCmd` and `basicOp` to execute once during each of the five time invervals *[t0, t0+100ms[, ..., [t0+400,t0+500ms[*. This behavior is in no way specified in the program described above. Similarly, the inputs of the system will all be consumed simultaneously at the beginning of the instant and the outputs of the system will all be produced simultaneously at the end of the instant (thus with a rate of 2Hz), which is not the expected behavior. To program this aspect correctly, primitives constraining the start and end dates of a node and the availability date of a flow are required. Finally, instantiating a node five times can be different from repeating it five times, if the node contains memories (`pre`). Five instances of the same node have five different memories, while five repetitions of the same node use the same memories. For instance, repeating a counter node five times obviously does not produce the same result as instantiating five different counters.

3.3 Synchronous languages in the development process

The different synchronous languages share the same execution model, however they do not all fit the same way in the development process. Synchronous languages can fall into two categories: *opened synchronous systems* or *closed synchronous systems.* In an opened synchronous system (all the equational synchronous languages), acquiring input on sensors and producing output on actuators is performed outside the synchronous program. The compiler generates code that needs to be completed by integration code. The integration code handles “communications” with the physical environment of the system (sensors and actuators) and activates the synchronous program. The activation rate of the program is determined by this integration code. We should point out that in these synchronous systems, the activation rate does not have to be periodic, even if it is the most natural assumption. Indeed the synchronous hypothesis simply holds as long as an instant does not start before the processes performed during the former instant have completed their execution, but the activation intervals can vary as long as this hypothesis is respected.
In a closed synchronous system (AAA methodology presented section 4.3 or SDF), acquiring inputs and producing outputs is performed by the synchronous program itself. The compiler generates a stand-alone executive, which does not require integration code. Consequently, the rate of the system is determined by the synchronous program itself. The next iteration starts as soon as the former iteration completes.

4. Real-Time Extensions to Synchronous Languages

We have seen that synchronous languages are well adapted for programming reactive systems but do not directly handle real-time constraints. These constraints must be taken into account manually by the programmer. Therefore, recent work aims at introducing real-time extensions in the synchronous model. The synchronous hypothesis is often qualified as the “zero execution time” hypothesis, because execution times are ignored. This does however not imply that time cannot be taken into account by the synchronous model. From a theoretical point of view, the duration of an instant is abstract. In practice, this duration is of course not null, and the synchronous hypothesis holds only if all the processes executing during one instant finish before the beginning of the next instant. The synchronous hypothesis does not prevent from considering the duration of an instant.

4.1 Implementing Lustre programs under real-time constraints

Recent work [8], proposed to introduce real-time aspects in Lustre through the use of *assumptions* made about the program environment and *requirements* about the program itself. Assumptions specify the base rate of the program (`basic_period=5`) as well as nodes execution times (`exec_time N in [3,4]`). Requirements constrain the availability date of a flow (`date(x)<5`) or the latency between two flows (`date(x)-date(y)>4`).

The compiler ensures, using static scheduling techniques, that if the assumptions hold, the requirements are satisfied. An additional primitive, `periodic_clock(k,p)` defines a clock of period *k* and of phase *p*. This clock is false during the *p-1* first instants and then true once every *k* instants. Such clocks are computable statically, which enables better scheduling analysis when opposed to classical Boolean clocks computed dynamically. These programs are compiled as a set of communicating tasks for a Time Triggered Architecture (TTA) [12].

Using these extensions, we first give in Figure 7 a corrected version of our case study programmed on a slow base rate. The assumptions specify the base rate of the program (500ms) and the durations of each node. The requirements constrain the availability of the inputs and outputs of the program.

```
node msu_slow(fromEnv0, fromEnv1, fromEnv2,
fromEnv3, fromEnv4: int;
              fromOtherMSU0, fromOtherMSU1,
fromOtherMSU2,
              fromOtherMSU3, fromOtherMSU4: int)
     returns (toEnv0, toEnv1, toEnv2, toEnv3,
toEnv4: int;
              toOtherMSU0, toOtherMSU1,
toOtherMSU2, toOtherMSU3,
              toOtherMSU4: int)
(hyp)
  basic_period=500;
  exec_time basic_op in [18, 20];
  exec_time applyCmd in [27, 30];
  exec_time downStream0 in [28, 30];
  exec_time downStream1 in [37, 40];
  exec_time upStream0 in [28, 30];
  exec_time upStream1 in [28, 30];
  exec_time upStream2 in [28, 30];
(req)
  date(fromEnv0)=0; date(fromOtherMSU0)=0;
  date(fromEnv1)=100; date(fromOtherMSU1)=100;
  date(fromEnv2)=200; date(fromOtherMSU2)=200;
  date(fromEnv3)=300; date(fromOtherMSU3)=300;
  date(fromEnv4)=400; date(fromOtherMSU4)=400;
  date(toEnv0)<=100; date(toOtherMSU0)<=100;
  date(toEnv1)<=200; date(toOtherMSU1)<=200;
  date(toEnv2)<=300; date(toOtherMSU2)<=300;
  date(toEnv3)<=400; date(toOtherMSU3)<=400;
  date(toEnv4)<=500; date(toOtherMSU4)<=500;
var us1, us2, ds: int;
    bop01, bop02,  bop11, bop12, bop21, bop22,
    bop31, bop32, bop41, bop42: int;
let
  ...
tel
```

Figure 7: Programming the case study in Lustre, with a slow base rate, using real-time extensions. Equations are the same as in Figure 6.

Compared to the previous Lustre program, this version constrains correctly the activation dates of the fast operations. However, the possible problems when instantiating a fast node with five different instances instead of repeating the same node five times remains. Furthermore, the handling of multi-rate aspects through the use of the `date` requirement is still quite heavy.

Consequently, we prefer a new version, programmed on the fast rate, given Figure 8. The base rate (100ms) is specified in the assumptions. The nodes of the fast task are executed once every five instants, using a periodic clock (`periodic_clock(5,5)`). The rest of the program remains the same as in Figure 4.

```
node msu_fast(fromEnv: int; fromOtherMSU: int)
     returns (toEnv, toOtherMSU: int)
(hyp)
  basic_period=100;
var bop1, bop02, bop12, bop22, bop32, bop42,
cur_ds, cur_us1: int;
    ds, us1, us2: int when periodic_clock(5, 5);
let
  clock5=periodic_clock(5, 5);
  bop32=0->pre(bop42);
  bop22=0->pre(bop32);
  bop12=0->pre(bop22);
  bop02=0->pre(bop12);
  bop1, bop42, toOtherMSU=basicOp(fromEnv,
fromOtherMSU, cur_ds);
  toEnv=applyCmd(cur_us1, bop1);
  us1, us2=upStream(bop02 when clock5,
                    bop12 when clock5,
                    bop22 when clock5,
                    bop32 when clock5,
                    bop42 when clock5);
  ds=downStream(us2);
  cur_ds=i_current(clock5, 0, pre(ds));
  cur_us1=i_current(clock5, 0, pre(us1));
tel
```

Figure 8: Programming our case study in Lustre, with a fast base rate, using real-time extensions.

According to the classical synchronous hypothesis, the slow nodes (`downStream` and `upStream`) must finish before the end of the (short) instant so we would encounter the same problems as mentioned previously. Therefore, this hypothesis is relaxed and processes must instead finish *before the end of their period*, which can be longer than one instant. This is possible because the program is compiled for a TTA platform. The Lustre code only initiates the executions of the tasks during the instant and the actual execution is handled by TTA tools (containing a preemptive scheduler). In this way, the computations performed by the Lustre program still end before the end of the instant, even if the computations performed by TTA tools do not. This new version seems to be quite adapted to our case study. However, some elements probably do not appear at the right place in the language. For instance, the node durations are specified inside the node where they are instantiated while it seems they would fit better at the top level of the program as these durations do not change for different instantiations. Communications between processes of different rates are also a little heavy to handle and could be more automated.

4.2 Preserving synchronous semantics under preemptive scheduling

The work presented in [16] does not extend synchronous languages directly but instead shows how the synchronous approach can be implemented on a multitask monoprocessor architecture, using preemption. This leverages the problem

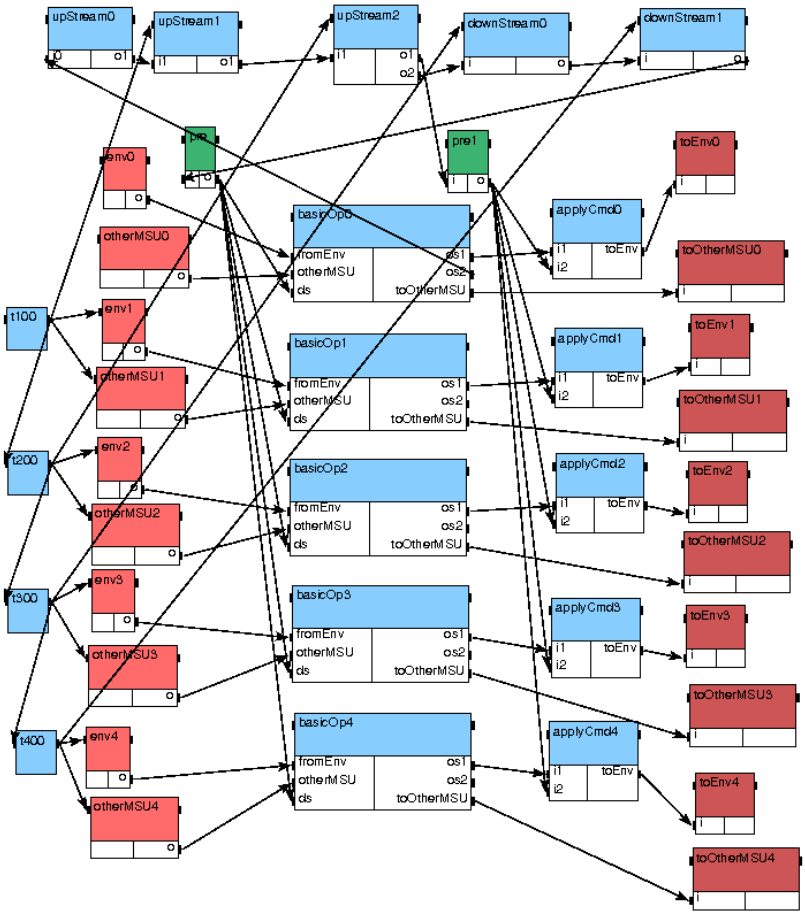

Figure 9: Programming our case study in SynDEx, with a fast base rate

encountered when programming with a fast base rate, namely the need for manually splitting slow operations so that they can be spread over several fast instants. Instead of splitting operations manually, this work allows preemption during nodes execution, while keeping the synchronous semantics. To this end, a specific communication protocol allowing communicating tasks to be preempted is defined (called Dynamic Buffering Protocol). This way, classical static scheduling algorithms like *rate-monotonic* can be used. The global compilation process consists in separating the synchronous program into several tasks with their associated priorities and release dates and then in scheduling these tasks using classical static algorithms and DBP for communications.

This solution is an interesting way to program our system using a fast base rate. However, the language itself remains unchanged. Consequently, the real-time aspects of the system do not appear clearly as such in the program, which can be not completely satisfying from a specification point of view. Furthermore, this enables less static analysis of the program and less compilation optimization.

4.3 The AAA methodology and the SynDEx software

The AAA methodology [10] (Algorithm Architecture Adequation) and its associated tool SynDEx are meant for designing distributed real-time embedded software. In this methodology, a system is modelled with graphs. The *algorithm graph* models the functional part of the system following the synchronous approach, while the *architecture graph* models the hardware of the system.

The algorithm graph is a Directed Acyclic Graph (DAG): vertices are operations (similar to nodes in Lustre) and edges stand for data communications. An operation can in turn be defined by an algorithm graph, enabling hierarchical definitions. The algorithm graph describes an iteration of the system repeated indefinitely. The architecture graph is a non-oriented graph: vertices are either *computation operators* or *communication operators* and edges stand for physical connections between those operators. Paths in the architecture graph strictly alternate between computation and communication operators. An architecture graph can model heterogeneous hardware including general purpose microprocessors as well as special purpose

integrated circuits. The user must finally specify the execution durations of operations on computing operators and the transmission durations of data types used by the operations on communication operators.
The Adequation process tries to find an optimized implementation of the algorithm graph on the architecture graph. It schedules and distributes the operations on computing operators and also schedules communications on communication operators. The result of the Adequation is then used to generate a synchronous distributed executive, which consists in one executive for each computing operator. The executive of each operator contains the code of the operations allocated on the operator but also code for synchronisations with other computing operators. Interesting work [11] has been done more recently to introduce multiple latency and periodicity constraints on operations, though the implementation currently does not seem to be complete, so we will not present this part in details.
Our case study can be programmed with SynDEx using a fast base rate very similarly to the version programmed in Lustre in Figure 5. We will rather detail a version using a slow base rate, which is inspired by a solution proposed by the authors of this methodology. The algorithm graph is given in Figure 9. Operations env_i and $otherMSU_i$ acquire inputs on sensors and operations $toEnv_i$ and $toOtherMSU_i$ produce outputs on actuators. These fast operations along with operations of the fast task ($basicOp_i$ and $applyCmd_i$) are repeated five times. The pre_i operations specify delayed communications, used for communications from slow operations to fast operations. The operations of the slow task are split into several operations, as previously in Lustre. We still need to constrain the fast operations to execute at the right time inside the slow cycle. As proposed by the team developing this methodology, this can be performed by introducing "timing operations". The specified execution duration of these operations is 100ms, i.e. one fast cycle. They are executed on a dedicated "timing operator". The first timing operation (`t100`) executes from date 0 to date 100, the second (`t200`) from 100 to 200 and so on. Finally, we add precedence dependences (dependences displayed on the top of operations) from the timing operations to the computing operations. As a consequence, the start date of operations can be constrained after a specific date. For instance the precedence from `t200` to `env2` delays the start date of operation `env2` after date 200 and transitively delays the start date of operation `basicOp2`. The architecture graph is a bi-operator architecture: the main computing operator and the timing operator linked by a "synchronisation" communication operator.
The major issue of this solution is that it does not constrain the end date of fast operations. Consequently the Adequation may schedule some fast operations too late. Still, the schedule is computed statically, so it can be checked (and modified by adding precedences to manually "guide" the Adequation) before actually implementing the system. Furthermore, if we obtain a correct schedule, the executive produced by SynDEx will be a correct implementation of our case study. Let us however point out that the implementation of the timing operator and timing operations may be difficult. It is very unlikely that an operator dedicated to timing purposes will be available on the final hardware. Yet, this operator could be emulated by a software process, for instance using threads (though this might not suit critical systems).

5. The Time-Triggered Approach

In the synchronous approach, order between processes is defined through their precedence relations. In the Time-Triggered approach [12], order between processes is defined through their start dates. We have seen previously that time-triggered architectures can be used as the execution platform of synchronous programs (section 4.1). In this section we will study how it can be used at the modelling level.

5.1 Giotto: a time-triggered language for embedded programming

Giotto [13] provides a programming model based on the time-triggered approach for developing embedded systems with hard real-time constraints. Such systems consist in a set of multi-periodic tasks and mode switching conditions for enabling or disabling tasks. Giotto assembles a set of tasks implemented outside Giotto (similarly to MetaH [14] or AADL [15]). The way operations are assembled abstracts from the execution platform, from the scheduling policy and from the implementation of the tasks.
The basic element in Giotto is the *task*. A task is defined by a set of input/output *ports*, by the function implementing the task (the external code) and by its frequency. The *instantiation* of a task specifies a *driver*, which provides the task inputs. A *mode* is made up of a set of tasks related by data dependences between task ports. A mode has a period and the frequency of a task is specified by the number of times it must be executed during the period of its mode. For instance, a task of frequency 2 in a mode of period 10ms is executed every 5ms. Data produced by this task is considered available only at the end of 5ms, even if the task executes in less than 5ms. Data dependences between tasks do not imply precedences. A task uses the last inputs produced before its start date. A *mode switch* from a mode *m* to a mode *m'* relies on a Boolean guard, which is evaluated at the frequency of mode *m*. If the

guard is true, *m'* becomes the active mode. Notice that the intersection of the set of tasks of *m* and *m'* does not have to be empty. A Giotto application consequently consists in a set of modes among which one is the initial mode. *Annotations* constrain the scheduling of the system. *Distribution annotations* constrain tasks to execute on a specific host. S*cheduling annotations* depend on the host type and on its scheduling policy (these annotations can be task priorities, deadlines and so on). Annotations (and scheduling algorithms) are not directly part of the global model of Giotto but instead depend on different possible implementations of Giotto.

5.2 Application to our case study

The Giotto program for our case study is given Figure 10. This time, the base rate of the program is the slow rate. The program is made up of one single mode (`mode standard()`) of period 500ms. During this period, the mode executes each fast operation five times (`taskfreq 5`), at regular time intervals and each slow operation one time (`taskfreq 1`). Actuators are also activated five times (`actfreq 5`), however it does not seem possible to repeat data acquisition on sensors. Communications between tasks are performed through drivers used at task instantiation.

```
output
  integer_port us1 :=integer_zero;
  integer_port us2 :=integer_zero;
  integer_port ds1 :=integer_zero;
  integer_port bop1 :=integer_zero;
  integer_port bop2 :=integer_zero;
  integer_port bop3 :=integer_zero;
  integer_port app1 :=integer_zero;

actuator integer_port toEnv;
         integer_port toMSU;

sensor integer_port fromEnv;
sensor integer_port fromMSU2;

////// tasks //////
task upStream(integer_port ius1) output (us1,
us2) state () {
  schedule UpStream(ius1,us1,us2)
}
task downStream(integer_port ids1) output (ds1)
state () {
  schedule DownStream(ids1,ds1)
}
task basicOp(integer_port iop1, integer_port
iop2, integer_port iop3)
    output (bop1,bop2,bop3) state () {
  schedule
BasicOp(iop1,iop2,iop3,bop1,bop2,bop3)
}
task applyCmd(integer_port iapp1, integer_port
iapp2)
    output (app1) state () {
  schedule ApplyCmd(iapp,app1)
}

////// drivers //////
driver prodEnvData(app1) output (integer_port
envData) {
  if constant_true() then
ComputeEnv(app1,envData)
}
driver prodMSUData(bop3) output () {
  if constant_true() then ComputeMSU(bop3,
msuData)
}
driver getUsData(bop1) output(integer_port
tous1) {
  if constant_true() then
    copy_integer1(bop1,tous1)
}
driver getDsData(us1) output(integer_port tods1)
{
  if constant_true() then
    copy_integer1(us1,tods1)
}
driver getBopData(fromEnv,fromMSU2,ds1)
    output (integer_port env, integer_port msu,
integer_port ds) {
  if constant_true() then
  copy_integer3(fromEnv,fromMSU2,ds1,env,msu,ds)
}
driver getAppData(bop2,us1)
    output(integer_port toiapp1,integer_port
toiapp2) {
  if constant_true() then
    copy_integer2(bop2,us1,toiapp1,toiapp1)
}

////// modes //////
start standard {
  mode standard() period 500 {
    actfreq 5 do toEnv(prodEnvData);
    actfreq 5 do toMSU(prodMSUData);
    taskfreq 1 do upStream(getUsData);
    taskfreq 1 do downStream(getDsData);
    taskfreq 5 do basicOp(getBopData);
    taskfreq 5 do applyCmd(getAppData);
  }
}
```

Figure 10: Programming our case study in Giotto

The main problem of this new version is that data dependences do not imply precedences, so we cannot constrain the execution order of tasks. We cannot specify delayed communications and cannot use hierarchical definitions either. Actually, the description level of Giotto is too high for our case study, it is meant for assembling high level tasks and we want to be able to describe our system more precisely.

6. Conclusion

We have presented different solutions for implementing embedded systems with real-time constraints, using high level programming languages. The synchronous approach is already frequently used in the embedded systems industry, at least for design or simulation purposes. It is indeed well adapted to the description of embedded systems. However we pointed out that the classical synchronous languages lack some high level primitives for programming real-time aspects. These aspects can be programmed manually, but the process is tedious and error-prone. The synchronous approach has been extended recently to tackle these difficulties, by adding real-time primitives or by changing the execution mechanisms of the synchronous programs. This makes for the closest solution to our problem but could still be improved, in particular when handling communications between processes of different rates. The time-triggered approach used at

modelling level, as implemented in Giotto, is not so well adapted to our problem. Time-triggered seems to fit better at the execution platform level (for instance to implement synchronous programs) instead of the programming level.
Through the different implementations proposed for our case study, we have emphasized a key issue: choosing the base rate on which to describe our system. Usually, the fastest rate of the system is chosen. However, less usual solutions using the slowest rate can be considered. Still, the approaches we studied all clearly favour one of the two solutions making the other one at least tedious if not impossible. We believe that designing a new language, which allows both could lead to improved expression capabilities and also to interesting compilation processes. Furthermore this would allow choosing an intermediate rate (neither the fastest nor the slowest) as the description rate of the system, which could sometimes be more adapted.

7. References

[1] A. Benveniste, G.Berry: *"The synchronous approach to reactive and real-time systems",* Readings in hardware/software co-design, p147-159, Kluwer Academic Publishers, 2002.

[2] F. Boussinot, R. De Simone: "*The Esterel Language"*, Proceedings of the IEEE, 79(9), 1991

[3] P. Raymond, D. Pillaud, N. Halbwachs: *"The synchronous data-flow programming language LUSTRE",* Proceedings of the IEEE, 79(9), 1991.

[4] A. Benveniste, P. Le Guernic, C. Jacquemot: "*Synchronous programming with events and relations : the SIGNAL language and its semantics*", Science of Computer Programming, 16: 103-149, 1991.

[5] M. Pouzet : "*Lucid Synchrone, version 3. Tutorial and reference manual*", Univertisy Paris-Sud, LRI, 2006.

[6] D.G. Messerschmitt, E.A. Lee: "*Synchronous Data Flow*", Proceedings of the IEEE, 75(9), 1987.

[7] Esterel Technologies, Inc: "*SCADE Language – Reference manual*".

[8] A. Curic: *"Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints",* PhD thesis, University Joseph Fourier, Grenoble, July 2005.

[9] H. Kopetz: *"Real-Time Systems: Design Principles for Distributed Embedded Application"*, Kluwer Academic, 1997.

[10] Y. Sorel, T. Grandpierre, C. Lavarenne: *"Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors"*, 7th International workshop on Hardware/Software Co-Design, CODES'99, Rome, Italy, 1999.

[11] L. Cucu, Y. Sorel: *"Real-Time scheduling for systems with precedence, periodicity and latency constraints"*, 10th international conference on Real-Time Systems (RTS'02), Paris, France, 2002.

[12] H. Kopetz: *"Real-Time Systems: Design Principles for Distributed Embedded Applications"*, Kluwer Academic, 1997.

[13] C. Kirsch, T. Henzinger, B. Horowitz: *"Giotto: A time-triggered language for embedded programming"*, Proceedings of the IEEE, 91:84-99, January 2003.

[14] P. Binns, P Vestal: *"Scheduling and communication in MetaH"*, Real-Time Systems Symposium, NC, USA, 1993.

[15] J. Hudak, P. Feiler: "*The architecture analysis & design language (aadl): an introduction"*, Technical Report, Carnegie Mellon University, 2006.

[16] C. Sofronis, S. Tripakis, P. Caspi: *"A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling"*, Proceedings of the 6th ACM & IEEE International conference on Embedded Software (EMSOFT '06), ACM Press, Seoul, Korea, 2006.

[17] MathWorks: *"Simulink: User's Guide"*