Simulation and Verification of Asynchronous Systems by means of a Synchronous Model[*]

Nicolas Halbwachs and Louis Mandel[†]
Vérimag[‡], Grenoble – France

Abstract

Synchrony and asynchrony are commonly opposed to each other. Now, in embedded applications, actual solutions are often situated in between, with synchronous processes composed in a partially asynchronous way. Examples of such intermediate solutions are GALS, quasi-synchronous periodic processes, deadline-driven task scheduling... In this paper, we illustrate the use of the synchronous paradigm to model and validate such partially asynchronous applications. We show that, through the use of sporadic activation of processes and simulation of non-determinism by the way of auxiliary inputs, the synchronous paradigm allows a precise control of asynchrony. The approach is illustrated on a real case study, proposed in the framework of the European Integrated project "Assert".

1 Introduction

It is well admitted, now, that the synchronous paradigm [4, 20] can significantly ease the modeling, programming, and validation of embedded systems and software. The synchronous parallel composition helps in structuring the model, without introducing non-determinism. The determinism of the model is also an invaluable advantage for its validation: tests are reproducible, and model-checking is not faced with the proliferation of states due to non-deterministic interleaving of processes.

It is also recognized that the synchronous paradigm is not the panacea, since it does not directly apply to intrinsically asynchronous situations, such as distributed systems, or applications mixing long tasks and urgent sporadic requests. This is why numerous works (see, e.g., [7] for a synthesis) are devoted to combining synchrony with asynchrony, or to extending the synchronous model towards less synchronous applications. For instance, "Communicating reactive processes" [11] or "Multiclock Esterel" [10] are extensions of the synchronous language Esterel [8] to cope with non perfectly synchronous concurrency. On the other hand, the paradigm of "Globally asynchronous, locally synchronous systems" (GALS) has been proposed [16, 1, 9] to describe general asynchronous systems, while keeping as much as possible the advantages of synchronous components. "Tag machines" [6, 5] are an even more general and abstract attempt in the same direction. [17] mixes synchronous (Signal) and asynchronous (Promela) models for verifying GALS.

Another track of research addresses the compilation of synchronous programs towards distributed or non strictly synchronous code. While some distribution methods aim at strictly preserving the synchronous semantics [13, 12], other proposals only preserve the functional semantics [14, 15, 27, 26].

Finally, other works concern the modeling of asynchronous systems within the synchronous paradigm. It is well-known since [24, 25] that a synchronous formalism can be used to express asynchrony. The only need is to express sporadic activation (or stuttering) of processes — which is allowed in all existing synchronous languages — and explicit non-determinism. The modeling tool Model-Build [2, 3] — developed within the European projects SafeAir and SafeAir2 — and the Polychrony workbench [23, 19, 18] are based on this idea. In this paper, we report on our use of this kind of approach in the framework of the Assert project.

Assert is a European Integrated Project devoted to the design of embedded systems from the system architecture level down to the code, with special emphasis on high-level modeling, proof-based design, and component reuse. Aerospace industry (avionics, launchers, and satellites) constitutes the main application domain of Assert. In this framework, we propose a methodology based on a high-level behavioral modeling and verification of an application, using a synchronous formalism. Since the automatic generation of distributed code is not an objective of the project, the automatic code generation is only applied separately to

[*]This work was partially supported by the European Commission under the Integrated Project Assert, IST 004033

[†]email: {Nicolas.Halbwachs, Louis.Mandel}@imag.fr

[‡]Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG associated with IMAG.

each software component of the model. Here, we report on our experience on modeling non synchronous applications using a synchronous formalism. The goal is to obtain an executable model of the whole application, allowing its early simulation and testing, together with the formal verification of its critical properties.

Such simulation of asynchronous systems within the synchronous model has obvious advantages: It allows an easy and precise description of the whole range of applications, from the pure synchronous to the completely asynchronous ones, but also all the intermediate solutions. Concerning GALS, the synchronous "islands" can be modeled by their actual code. The model is naturally executable, for early simulation and testing. As for synchronous programs, properties can be expressed by observers in the same formalism, since the basic time scale is available for observing the behavior of the processes exactly. These observers can be used for verification and automatic testing. In the case of verification by model-checking, the state explosion due to non-deterministic interleaving is limited.

In the paper we will not use a particular formalism, but rather an abstract notion of synchronous machines (Section 2) in the style of [21]. In Section 3, we explain how we express asynchrony in this framework, by allowing sporadic activation, and "simulating" non-determinism by means of auxiliary inputs. As an illustration, we use a case study which is briefly presented in Section 4. The case study is derived from an actual component of the "Automatic Transfer Vehicle", a spacecraft in charge of supplying the International Space Station. This application contains two kinds of non synchronous features:

- First it consists of two redundant components running on so-called "quasi-synchronous" processors: such processors are supposed to run with "almost" consistent clocks, in the sense that their maximum instantaneous drift is precisely bounded.
- Second, each component involves three tasks, the first two being periodic with different periods, and the third one being sporadic and urgent.

Section 5 is devoted to the modeling of quasi-synchrony: we show how to express the quasi-synchronous assumption about the processors'clocks, and how to describe a scheduler satisfying this assumption in our synchronous formalism. The approach is applied to the case study and allows some distributed properties to be automatically verified. Then the modeling of multitasking is presented in Section 6, and applied to the case study, where is allows to prove the functional determinism of the system, in spite of the non-deterministic scheduling.

2 The synchronous paradigm

In this section, we recall the only necessary features about synchronous programming which are of interest for our modeling activity. Externally, a synchronous program behaves as a sequence of atomic steps, which can be periodic or sporadic, according to the way the program is activated. To perform one step of the program, the environment has to provide it with its current inputs; the step itself consists in computing the current outputs, as a function of the current inputs and the current internal state of the program (which generally has remanent variables encoding this state), and of updating the state for the next step.

The specific feature of synchronous programs is the way internal components behave with each other: when several components are composed in synchronous parallelism, one step of the whole composition consists of a "simultaneous" step of all the components, which can communicate with each other during the execution of the step. This execution is guaranteed to be *deterministic*, a very important property of synchronous programs, since it makes much easier the understanding of programs, as well as their testing and verification.

To be more precise, and following the presentation of [21], a synchronous component is a straightforward generalization of synchronous circuits (Mealy machines) to work with arbitrary data-types: such a machine has a memory (a state), and a combinational part, computing the output and the next state as a function of the current input and the current state.

For instance, Fig. 1.a pictures a machine with two inputs, x and y, one output z and one state variable s, and one can define a step of the machine by the functions, say f_o and f_s, respectively giving the output and the next state from the current inputs and the current state:

$$z = f_o(x, y, s) \quad , \quad s' = f_s(x, y, s)$$

The behavior of the machine is the following: it starts in some initial state s_0. In a given state s, it deterministically reacts to an input valuation (x, y) by returning the output $z = f_o(x, y, s)$ and by updating its state into $s' = f_s(x, y, s)$ for the next reaction. When the set of states (the domain of s) is finite, we will sometimes specify such a machine as an input/output automaton, whose states are the possible values of s, and where there is a transition from s_1 to s_2, labeled by i/o, if and only if $s_2 = f_s(i, s_1)$ and $o = f_o(i, s_1)$.

These machines can be composed in parallel, with possible "plugging" of one's outputs into the other's inputs (Fig. 1.b), as long as these wirings don't introduce any combinational loop. Such a composition is shown by Fig. 1.b, where the variables are defined by

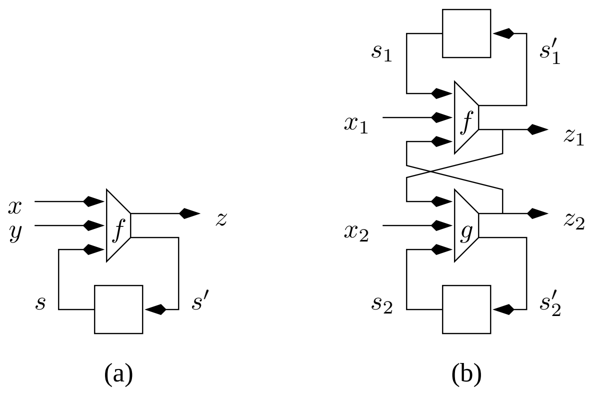

Figure 1. Synchronous machines and their composition

$$z_1 = f_o(x_1, z_2, s_1)\ ,\ z_2 = g_o(x_2, z_1, s_2)$$
$$s'_1 = f_s(x_1, z_2, s_1)\ ,\ s'_2 = g_s(x_2, z_1, s_2)$$

and where either the results of $f_o(x_1, z_2, s_1)$ should not depend on z_2, or the results of $g_o(x_2, z_1, s_2)$ should not depend on z_1, to avoid combinational loops.

To conclude this section, let us give two very simple examples of synchronous machines that we will use later. The first one is a single "delay" machine (the "pre" operator of Lustre): the machine δ receives an input i of some type τ, and returns its input delayed by 1 step; it has a state variable s of type τ and is defined by

$$f_o(i, s) = s \quad , \quad f_s(i, s) = i$$

Our second example is a *sampler* $\beta(b)$, with an input i of type τ and a Boolean input b, which returns the value of i when b is true, or its previous output when b is false (it would be written "current(i when b)" in Lustre). It is defined by

$$f_o(i, s) = f_s(i, s) = \text{if } b \text{ then } i \text{ else } s$$

3 Expressing asynchrony in the synchronous model

Basically, the difference between synchrony and asynchrony is that, in the synchronous model, each significant "event" is precisely dated with respect to other events and with respect to the sequence of steps. As soon as synchrony is released, the date of some events becomes unknown, or not precisely known, meaning that the temporal behavior becomes non-deterministic.

According to our presentation of the synchronous model, to express asynchrony, we need, on one hand, that components don't necessarily participate in all steps, and on the other hand, to express non-determinism (which is on purpose forbidden in synchronous languages!).

3.1 Sporadic activation: "activation condition" vs. "clock enable"

All synchronous languages propose some ways of preventing a component from reacting (the "suspend" statement of Esterel, or the "clock" mechanism of Lustre and Signal). However, we don't want to bother about "absent" values or signals, induced by these notions. In Scade, the notion of "activation condition" allows a node to be activated sporadically, its output and state keeping their previous values when the activation condition is false; so, an additional memory is implicitly requested to record the previous output. We can also consider a more basic — while probably less intuitive — sporadic activation, inspired by the behavior of circuits submitted to a "clock enable": when the condition is false (or 0), only the machine memory gets frozen, and the combinational part continues computing outputs. More precisely, if M is a synchronous machine, with input i, output o, and state s, defined by the functions f_o and f_s:

- the *clocked activation* of M by b, noted $M \triangleleft b$, is the machine obtained by adding a new Boolean input b to M, and whose output and state functions f'_o and f'_s are as follows:
$$f'_o((i, b), s) = f_o(i, s)$$
$$f'_s((i, b), s) = \begin{cases} s & \text{if } b = 0 \\ f_s(i, s) & \text{if } b = 1 \end{cases}$$
- the *conditional activation* of M by b, noted $M \blacktriangleleft b$, has also b as a new input, its state is a pair (s, o^-), and its output and state functions f'_o and f'_s are as follows:
$$f'_o((i, b), (s, o^-)) = \begin{cases} o^- & \text{if } b = 0 \\ f_o(i, s) & \text{if } b = 1 \end{cases}$$
$$f'_s((i, b), (s, o^-)) = \begin{cases} (s, o^-) & \text{if } b = 0 \\ (f_s(i, s), f_o(i, s)) & \text{if } b = 1 \end{cases}$$

The conditional activation $M \blacktriangleleft b$ can be built from the clocked activation, by plugging the output of $M \triangleleft b$ into a sampler $\beta(b)$ (Fig. 2.a). Graphically, we shall note a "clock enable" $M \triangleleft b$ (resp., an "activation condition" $M \blacktriangleleft b$) as a white (resp., black) arrow input on top of the machine it controls (Fig 2).

3.2 Non-determinism

We will classically model non-determinism by means of additional inputs — often called "oracles" — to the model.

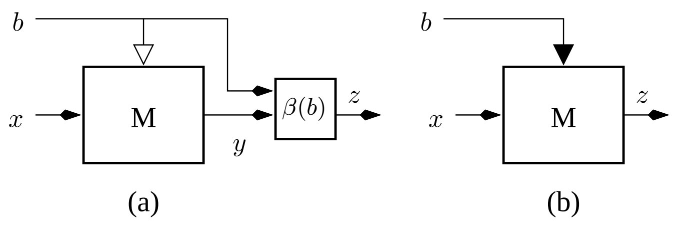

Figure 2. Modeling an "activation condition" with a "clock enable"

These oracles will be used to control non-deterministic choices. This way of expressing non-determinism has some advantages over built-in non-deterministic constructs of many specification languages:

- On one hand, the non-determinism is clearly localized and controlled: one can replay the same execution twice, just by providing the same oracles.
- On the other hand, the non-determinism can be reduced, by imposing some constraints on oracles. We will make an intensive use of this feature, in particular to express known scheduling constraints.

3.3 General principles

By combining sporadic activation and oracle-driven non-determinism, we are able to express any non-synchronous composition of synchronous processes. The general construction is the following (see Fig. 3.a): the processes are all sporadically activated according to activation conditions emitted by a global scheduler. The scheduler is non-deterministic: it receives one Boolean oracle for each condition it has to elaborate. But it can restrict this complete non-determinism by enforcing constraints among the conditions it actually emits towards the processes.

For instance, in the example of Fig 3.a, the scheduler could prevent the processes from being activated simultaneously, by setting

$$C_P = \Omega_P \ , \ C_Q = \Omega_Q \wedge \neg C_P$$

This would simulate two asynchronous processes, communicating through a shared memory: since, when P is not activated, its outputs keep their last values, then Q gets these last written values just as it would read them in a shared memory. More complex communication mechanisms can be modeled as communication processes, which can need their own activation conditions (see Fig 3.b and Section 6).

The modeling of a new composition mechanism then consists in expressing constraints on activation conditions, to be inserted in the scheduler. Since this task is quite difficult and error-prone, we would like to define, once and for all, a library of such mechanisms to be simply instantiated by the user, using a simple formalism to be defined.

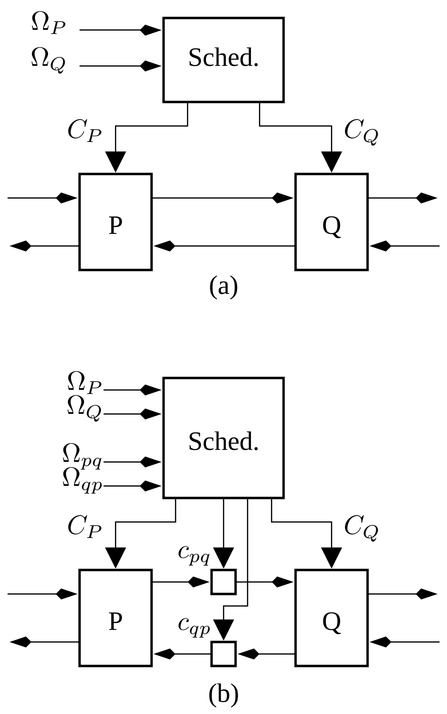

Figure 3. Modeling non synchronous executions

4 The case study

The PFS ("Proximity Flight Safety") case study was proposed by EADS Space Transportation, and concerns an equipment of the "Automatic Transfer Vehicle" (ATV) in charge of supplying the International Space Station (ISS). The role of the PFS is to ensure the safety of the approach of the ATV to the ISS: when anything goes wrong, the PFS is in charge of performing a "collision avoidance maneuver" (CAM), i.e., to safely move the ATV apart from the ISS, and to orient it towards the sun, waiting for new instructions.

The CAM is performed by two redundant units, called "Monitoring and Safing Units" (MSU), running on two computers. At each instant, one of them is the master, but can resign this role if it detects its own failure, in which case the other MSU becomes the master. However, the master may not change when a CAM is in progress. Once an MSU has given up its mastership, it never recovers.

Each MSU:

- detects anomalies, which can be failures of the main

computer (or "fault-tolerant computer pool", FTCP), abnormal state of the bus, erroneous position or speed of the ATV, "red button" pressed from inside the ISS;
- detects its own failures, which can involve a master change;
- is able to perform a CAM

The whole system is supposed to tolerate at most two faults: the one which gives raise to a CAM, and a fault of one MSU. As a consequence, one doesn't have to consider the case of two faulty MSUs.

The actual system was modified, in the framework of the Assert project, in order to introduce more asynchronous aspects: not only are the two MSUs running on separate, so-called "quasi-synchronous" computers (see below), but each of them is supposed to be made of three tasks, activated at different rates: two periodic tasks, with different periods, and a sporadic, event-driven task. Each MSU is described by a Scade model, which can be viewed as a synchronous machine.

In the following sections, we consider the modeling of quasi-synchrony and multitasking.

5 Quasi-synchrony

Quasi-synchrony is a very common composition mechanism for periodic processes. It has been formalized by Paul Caspi in [14]. It consists in executing several periodic processes on different processors, supposed to run with the same clock. Now, instead of ensuring the exact synchronization of clocks, using costly mechanisms, one prefers to tolerate a small clock drift, as long as the following property holds:

> Between any two successive activations of one periodic process, the process on any other processor is activated either 0, or 1, or at most 2 times.

In other words, in the case of two processes, the possible interleaving of activations of the processes can be the one pictured in Fig. 4. Notice that the difference between the absolute numbers of activations on each processor can increase unboundedly (e.g., one process can be activated twice more often than the other), so it is not a very strong assumption. The important property of quasi-synchronous composition is that, if the processes communicate with each other simply by shared memory, each of them is guaranteed to miss at most one sample of the other's output in a row, and conversely, to "duplicate" (i.e., read twice the same) at most twice one sample of the other's output. Fig. 5 illustrate these cases of imperfect communication: for instance, the sample 1 of P_1's output is read twice by P_2, while the sample 1 of P_2's output is missed by P_1.

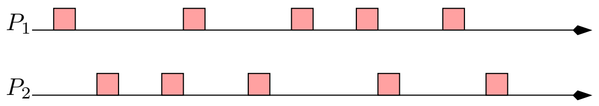

Figure 4. Quasi-synchronous interleaving

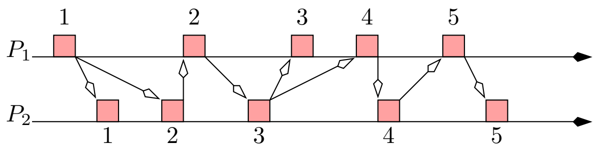

Figure 5. Quasi-synchronous communication

5.1 Synchronous modeling of quasi-synchrony

For modeling the quasi-synchronous composition of two processes P and Q, we have to activate them with two conditions C_P and C_Q, such that

> between two occurrences of C_P there are at most 2 occurrences of C_Q, and conversely, between two occurrences of C_P there are at most 2 occurrences of C_Q.

This informal specification contains some ambiguities, mainly concerning the case where both conditions are true at the same time. For the sake of generality, we consider that these simultaneous occurrences are possible. Our interpretation of quasi-synchrony will be the following:

> Each condition cannot be true *alone* more than twice, without the other being true meanwhile; if a condition occurs twice alone in a row, the other condition must follow alone.

We can represent this specification by an automaton producing the "language" of correct behaviors: a "letter" in this language is a Boolean monomial on the values of the conditions, namely an element of

$$\{C_P.C_Q \ , \ C_P.\overline{C_Q} \ , \ \overline{C_P}.C_Q \ , \ \overline{C_P}.\overline{C_Q}\}$$

The states of the automaton represent the "advance" of one condition over the other, i.e., the number of times it has been true alone since the last time the other was true. So, we get 5 states:
- $1P$: C_P has been true alone once since the last occurrence of C_Q
- $2P$: C_P has been true alone twice since the last occurrence of C_Q
- conversely for $1Q$ and $2Q$

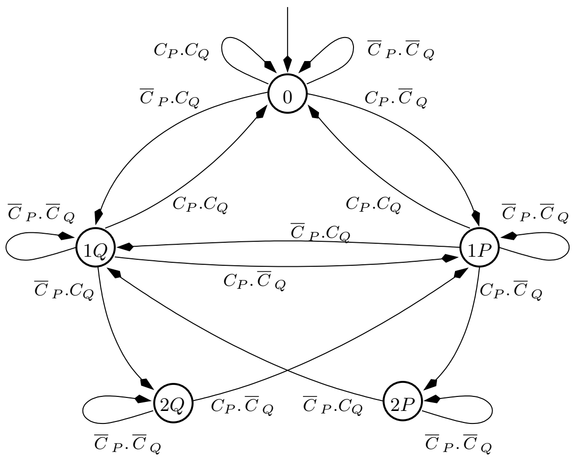

Figure 6. Acceptor of quasi-synchronous activations

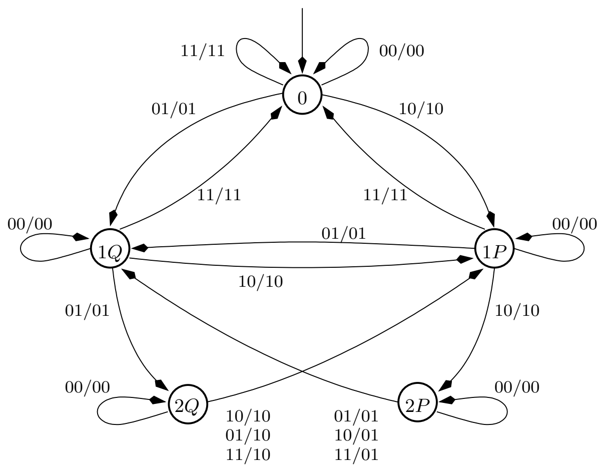

Figure 7. Scheduler for quasi-synchrony

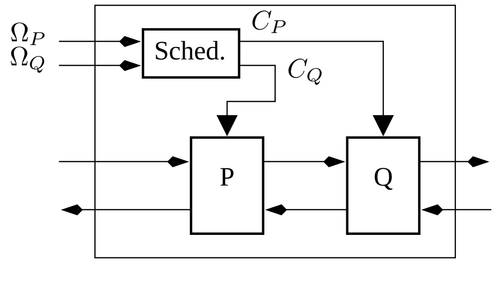

Figure 8. Quasi-synchronous activation

- 0 : the initial state, and the one which is reached whenever both conditions are simultaneously true, since, in such situation, none of them is "in advance".

The corresponding automaton is represented in Fig 6. This automaton can easily be transformed into a scheduler (Fig. 7), receiving the oracles (Ω_P, Ω_Q), and computing the activation conditions (C_P, C_Q): in states $0, 1P, 1Q$, it just copies its inputs, while in states $2P$, and $2Q$ it transforms forbidden inputs into (arbitrary) correct ones.

Now, the quasi-synchronous composition of P and Q consists just in composing in parallel this scheduler with $P \blacktriangleleft C_P$ and $Q \blacktriangleleft C_Q$ (Fig. 8).

5.2 Application to the case study

We used such a model in Lustre, first to perform extensive simulations, and then to apply the Lesar model-checker [22] to verify some properties of the PFS case study:

1. It was immediate to verify that at each instant, at most one MSU is the master.

2. Now, when we try to prove the converse, i.e., that there is one master at each instant, the verification fails and provide an obvious counterexample: of course, since the communication is not instantaneous, when a MSU gives up its role of master there is an interval of time before the other takes the mastership. This short interval was taken into account by the designers of the PFS, and considered acceptable. However, we were able to verify a weaker property: if we call C_1, C_2 the activation conditions of the two MSUs, and if the first MSU gives up its role of master at some tick of C_1, then at the tick of C_2 following the next tick of C_1, the second MSU is the master.

Notice that both properties only hold because, when an MSU gives up the master role, it never changes its mind later on; this implies that its decision is necessarily maintained for more than one step, so it is perceived by the other MSU.

6 Multi-tasking

As a second example, we consider a process Q which is activated on a multiple of the basic period of a main, priority process P. Q is activated once every n activations of P, and its execution can spread over n cycles of P, thanks to some preemptive scheduler. However, it is guaranteed to fit in its period, i.e., to terminate within at most n cycles of the basic period. A possible scheduling of the computations is shown in Fig. 9, with $n = 3$.

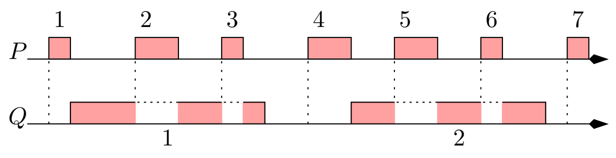

Figure 9. Multitasking

This is a quite common situation of multi-tasking, which happens, e.g., in the PFS case study. It could be generalized, for instance, by imposing to Q a deadline different from its period, or even by considering non periodic activations of Q.

Here, we are faced with the problem of modeling tasks with *durations*, which seems contradictory with the synchronous abstraction. The obvious solution is just to consider that the outputs of Q are available later than its activation. So, Q will be activated at some condition C_Q, on which its inputs will be sampled, and its outputs will be available according to another condition A_Q. Notice that this modeling still supposes that Q samples its inputs synchronously, and deposits its outputs also synchronously. On one hand, releasing this assumption would make the model much more complex, and on the other hand, a design which would not respect this condition would be questionable.

6.1 Synchronous modeling of multi-tasking

As said before, a sporadic non instantaneous process Q will be modeled by means of two conditions: an activation condition C_Q, which triggers the atomic sampling of its inputs, and an availability condition A_Q which corresponds to the simultaneous availability of its outputs. In the case addressed here (which occurs in the PFS case study), the activation condition occurs every n cycles of a more priority process P, and the outputs are supposed to be available at some instant during these n cycles. In other words, if Q is activated at cycle k, it will deliver its outputs either at cycle k, or $k+1$, or..., or $k+n-1$ at the latest.

So, we have two tasks:

- To define C_Q, we need an operator which "divides" the activation C_P of P by the parameter n. Notice that this operator is deterministic: C_Q should happen simultaneously with the $1st$, $(n+1)$th, $(2n+1)$th, ..., occurrences of C_P.
- To define A_Q, we need to express the condition that it happens exactly once (non-deterministically) inside each interval starting at an occurrence of C_Q and ending just before the next occurrence of C_Q.

Definition of the activation: The construction of C_Q is deterministic, so C_Q is not an oracle restricted by an assertion, but the result of an operator, building it from C_P and n. Obviously, the construction of C_Q involves a counting of occurrences of C_P. A machine $divide(n)$ taking C_P as input and computing C_Q can be defined by the very simple automaton of Fig. 10.

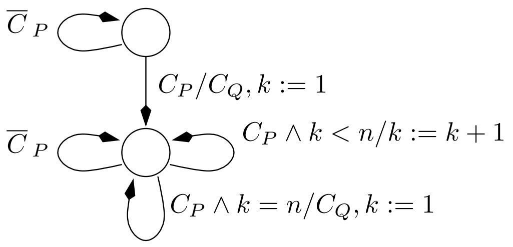

Figure 10. "Division" of an activation condition

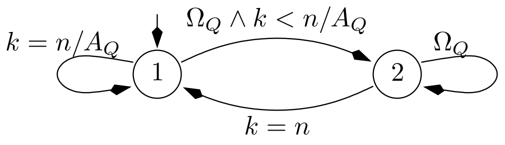

Figure 11. Signaling the end of the task

Definition of the availability condition: The availability condition is non-deterministic. It will be defined using an oracle, say Ω_Q: using the same counter k as for the division, in each interval starting when k is set to 1, and ending when k reaches $n-1$, the first occurrence of Ω_Q is considered as an occurrence of A_Q. If Ω_Q does not occur during the whole interval, A_Q is sent at the last instant.

The automaton of Fig 11, supposedly clocked by C_P, does the job: in State 1, if Ω_Q happens before the end of the interval (i.e., when $k < n$), A_Q is emitted, and the automaton moves to State 2 where it ignores subsequent occurrences of Ω_Q until the end of the current interval; if the end of the interval (i.e., $k = n$) occurs in State 1, A_Q is emitted, if it occurs in State 2, the automaton moves back to State 1.

Now, the whole model can be built as in Fig. 13: The scheduler (Fig. 12) is the product of the automata of Fig. 10 and 11: it receives C_P and the oracle Ω_Q and computes C_Q and A_Q. C_P and C_Q are the respective activation conditions of P and Q, and A_Q is used to delay the outputs of Q by means of samplers $\beta(A_Q)$. Q is activated by C_Q, computed from C_P, and the availability condition A_Q is used to sample all outputs of Q.

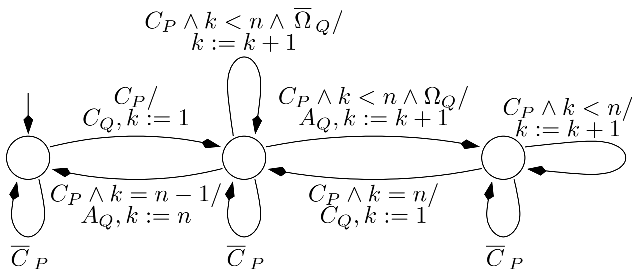

Figure 12. The whole scheduler

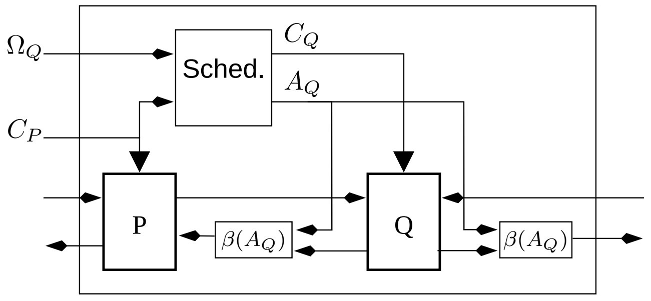

Figure 13. Model of multi-tasking

6.2 Application to the case study

The PFS case study presents such a situation of multitasking, together with a purely sporadic task (i.e., non periodic) which was just modeled with an activation condition. On our model of this design, the verification of the properties considered in §5.2 did not raise any problem.

Now, interestingly, concerning the periodic tasks, the designers of the case study were aware of the risk of functional non-determinism involved by the multitasking. This non-determinism was avoided by taking into account the results of the "slow" task at the latest: the results of the task executed in some interval are only acquired by the "fast" task at the beginning of the next interval[1] (see Fig. 14), so the non-deterministic termination date of the slow task has no functional consequence. Notice that this is precisely the condition assumed by [27] to generate deterministic multitask code from a synchronous program. We were able to verify this functional determinism with the Lesar model-checker, just by comparing the outputs of our model with the ones of a purely synchronous version of the PFS (where the slow task is completely executed at the beginning of the interval, i.e., $A_Q = C_Q$).

[1]With the previous notations, it means that the fast task P does not access directly the outputs o of the slow task Q, but reads instead $\beta(C_Q)(\delta(o))$, i.e., the sampling on C_Q of the delayed version of o.

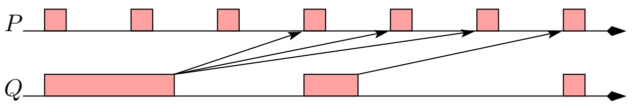

Figure 14. Data communication between tasks in the PFS

7 Conclusion

The modeling technique described in this paper is not new. It is routinely used by many people in the community of synchronous languages. For instance, it was used in a very similar way by [15, 27] to validate the code generation towards quasi-synchronous architectures and multitasking implementations. A very similar approach is also systematized in the Model-build toolbox [2, 3]. It is also the way [9] simulates Multi-clock Esterel in standard Esterel. However, it is also a solution which is ignored by many people *outside* the community of synchronous languages, who consider that synchrony only works for centralized, sequential implementations. This is why we wanted to popularize it in a wider community, taking advantage of a very good case study provided in the Assert project.

The advantages of this modeling approach have been indicated before: it provides a quite systematic way of modeling asynchronous *or partially asynchronous* behaviors, and to cleanly merge these behaviours with synchronous parts; it provides an executable model, on which many tools can be directly used, for simulation, testing, formal verification, debugging... Moreover it allows a significant reduction of the validation cost, since the synchronous description of the synchronous parts generally involves a drastic reduction of the number of states.

In this paper, we chose to present the approach independently of any specific language. Now, the use of a real language obviously makes the work easier: the real case study was treated with Scade and Lustre, which provide convenient features, like process abstraction (nodes) and parameterization, assertions to constrain the behaviors of oracles, and allow actual Scade models of the synchronous parts to be directly imported.

It remains to provide a library of standard components (scheduling constraints, communication modules, ...) in Scade, to alleviate the task of the user. It is one of our objectives in the Assert project.

References

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis:

An integrated electronic system design environment. *Computer — IEEE Computer Society,*, 36(4):45–52, 2003.

[2] P. Baufreton. SACRES: A step ahead in the development of critical avionics applications. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control: Second International Workshop, HSCC'99*. LNCS 1569, Springer-Verlag, 1999.

[3] P. Baufreton. Visual notations based on synchronous languages for dynamic validation of gals systems. In *CCCT'04 Computing, Communications and Control Technologies*, Austin (Texas), Aug. 2004.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sept. 1991.

[5] A. Benveniste, B. Caillaud, L. Carloni, and A. Sangiovanni-Vincentelli. Tag machines. In *EMSOFT'2005*, Sept. 2005.

[6] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling: capturing causality and the correctness of the loosely time-triggered architectures. In G. Buttazzo and S. Edwards, editors, *4th Int. Conf. on Embedded Software, EMSOFT'04*, Sept. 2004.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), Jan. 2003.

[8] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] G. Berry and E. Sentovich. Embedding synchronous circuits in GALS-based systems. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, Oct. 1998.

[10] G. Berry and E. Sentovich. Multiclock Esterel. In *Correct Hardware Design and Verification Methods, CHARME'01*, Livingston (Scotland), Sept. 2001. LNCS 2144, Springer Verlag.

[11] G. Berry, R. K. Shyamasundar, and S. Ramesh. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, POPL'93*, Charleston, Virginia, 1993.

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *LCTES 2003*, San Diego, CA, June 2003.

[13] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999. Research report INRIA 3491.

[14] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems, the quasi-synchronous approach. In *SAFECOMP'01*. LNCS 2187, 2001.

[15] P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In *FTRTFT'2000*, Pune, India, Sept. 2000. LNCS 1926.

[16] D. Chapiro. Globally-asynchronous locally-synchronous systems. Phd thesis, Stanford University, 1984.

[17] F. Doucet, M. Menarini, I. H. Krüger, R. Gupta, and J.-P. Talpin. A verification approach for GALS integration of synchronous components. In *FMGALS'2005*, Verona (Italy), July 2005.

[18] A. Gamatié and T. Gautier. The signal approach to the design of system architectures. In *10th IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2003)*, pages 80–88, Huntsville (Alabama), Apr. 2003.

[19] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the signal language. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'2003)*, pages 144–151, Toronto, May 2003.

[20] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[21] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*. LNCS 2491, Springer Verlag, Oct. 2002.

[22] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, Sept. 1992.

[23] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, Apr. 2003.

[24] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[25] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

[26] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *5th Int. Conf. on Applications of Concurrency in System Design, ACSD'05*, Saint-Malo (France), 2005.

[27] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, June 2004.