Necessary and Sufficient Conditions for Deterministic Desynchronization

Dumitru Potop-Butucaru
INRIA, Rocquencourt
France
dumitru.potop@inria.fr

Robert de Simone
INRIA, Sophia Antipolis
France
rs@sophia.inria.fr

Yves Sorel
INRIA, Rocquencourt
France
yves.sorel@inria.fr

ABSTRACT

Synchronous reactive formalisms associate concurrent behaviors to precise schedules on global clock(s). This allows a non-ambiguous notion of "absent" signal, which can be reacted upon. But in desynchronized (possibly distributed) implementations, absent values must be explicitely exchanged, unless behaviors were already provably independent and asynchronous (a property formerly introduced as endochrony).

We provide further criteria restricting "reaction to absence" to allow correct desynchronized implementation. We also show that these criteria not only depend on the desired correctness properties, but also on the desired structure of the implementation.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-Time and Embedded Systems D.3.1 [Formal Definitions and Theory]: Semantics

General Terms: Theory, Languages, Design

Keywords: Desynchronization, GALS, Endochrony, Execution machine, Correctness, Determinism, Reaction to signal absence, Kahn process networks

1. INTRODUCTION

Synchronous reactive formalisms [9, 4] are modeling and programming languages used in the specification and analysis of safety-critical embedded systems. They comprise (synchronous) concurrency features, and are based on the Mealy machine paradigm: Input signals can occur from the environment, possibly simultaneously, at the pace of a given *global clock*. Output signals and state changes are then computed before the next clock tick, grouped as one *atomic reaction*. Because common computation instants are well-defined, so is the notion of signal *absence* at a given instant. Reaction to absence is allowed, *i.e.*, a change can be caused by the absence of a signal on a new clock tick. Since component inputs may become local signals in a larger concurrent system, *absent* values may have to be computed and propagated, to implement correctly the synchronous semantics.

When an asynchronous implementation is meant, where possibly distributed components communicate via message passing, the explicit propagatation of all *absent* values may clog the system to a certain extent. A natural question arises: *when can one dispose of such "absent signal" communications?*

Sufficient conditions, known as *(weak) endochrony* [3, 8, 14, 13], have been introduced in the past to figure when the *absent* values can be replaced in the implementation by actual absence of messages without affecting its *correctness* and *determinism*. Weak endochrony determines that compound reactions that are apparently synchronous can be split into independent smaller reactions that are asynchronously feasible in a confluent way (one after the other instead of simultaneously), so that the first one does not discard the second. This is also linked to the Kahn principles for networks [11], where only internal choice is allowed to ensure that overall lack of confluence cannot be caused by input signal speed variations.

In this paper, we rephrase these issues to better show their mutual relations, we strengthen the theory by asserting *necessary and sufficient conditions*, and we show that these conditions need to take into account the structure of the *execution machines* used to give a globally asynchronous implementation to a synchronous specification.

Outline. Section 2 explains what we understand by synchronous specification, asynchronous implementation, and signal absence. It starts with a brief introduction to Kahn process networks, which includes the formal notations for the asynchronous framework. Section 3 covers the representation of signal absence in various languages. Section 4 is on reaction to signal absence. It gives its formal definition, implementation details, and examples. Section 5 takes into account concurrency and gives our main result. We give examples in Section 6 and conclude in Section 7.

2. BASIC NOTIONS

2.1 Kahn process networks

In 1974, Gilles Kahn wrote his seminal paper [11] on what is known today as *Kahn process networks (KPN)*. He introduced a simple language for defining distributed systems of communicating sequential processes, and fully specified the underlying communication and execution mechanisms.

In a KPN, interprocess communication is done through *message passing* along *channels* (asynchronous lossless FIFOs). When reading a channel, a process is blocked until a message is available. There is no block on writing. Once sent, a message reaches its destination in a finite (but unbounded) time, meaning that one communication cannot block another indefinitely. On the receiver end, the message remains on the channel until it is read. Any number of messages can be sent before one is read (the channels are unbounded). No other communication or synchronization mecha-

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

nism exists between processes (which run in parallel). In particular, time is not used to make decisions or trigger computations.

The simple model of the KPN proved to be an excellent basis for the *compositional* modelling of deterministic systems that operate infinitely using limited resources [12]. Variants or extensions of the original model are currently used in a variety of industrial and academic settings.

2.1.1 Formalization

The formal analysis of a KPN is based on the representation of each process as a *stream function* converting *input histories* into *output histories*. Formally, a Kahn process network has a set of FIFO channels C and a set of processes P.

Given a channel $c \in C$, its *domain* $\mathcal{D}_c$ is the set of values that can be transmitted as messages over c. We denote with ${\mathcal{D}_c}^*$ the set of all finite sequences over $\mathcal{D}_c$, and with ${\mathcal{D}_c}^\omega$ the set of all finite or infinite sequences. Given an execution of the KPN, the *history* $Hist(c)$ of a channel c is the sequence of all messages that transit c during the execution. For a finite execution, $Hist(c) \in {\mathcal{D}_c}^*$. For an infinite execution, $Hist(c) \in {\mathcal{D}_c}^\omega$. The set ${\mathcal{D}_c}^\omega$ is ordered by the prefix order $\preceq$. We denote with ϵ the empty sequence. Any increasing sequence $(h_j)_{j=1}^{\infty}$ in ${\mathcal{D}_c}^\omega$ has a limit $\lim_{j\to\infty} h_j$. The prefix order and the limit operator on individual ${\mathcal{D}_c}^\omega$ induce a product order and a limit operator on any product set $\prod_{j=1}^{n} {\mathcal{D}_{c_j}}^\omega$.

Each process $f \in P$ has zero or more input channels $ci_j \in C, 1 \leq j \leq n$, and zero or more output channels $co_j \in C, 1 \leq j \leq m$. Given an execution of f, the *input history of* f is $(Hist(ci_j))_{j=1}^{n} \in \prod_{j=1}^{n} {\mathcal{D}_{ci_j}}^\omega$ and the *output history of* f is $(Hist(co_j))_{j=1}^{m} \in \prod_{j=1}^{m} {\mathcal{D}_{co_j}}^\omega$.

Given that f is *sequential* and *deterministic*, its behavior can be defined as a *function* from input histories to output histories:

$$f : \prod_{j=1}^{n} {\mathcal{D}_{ci_j}}^\omega \to \prod_{j=1}^{m} {\mathcal{D}_{co_j}}^\omega$$

We shall call this function the *stream function* of f, and denote it with the process name.

2.1.2 The Kahn principle

All the stream functions associated with Kahn processes are:

- *Monotonous*, in the sense of $\preceq$, meaning that giving more messages on the input channels results in more output messages.
- *Continuous*, in the sense of the limit operator, meaning that for any sequence of input histories $h_k \in \prod_{j=1}^{n} {\mathcal{D}_{ci_j}}^\omega, k \geq 1$ such that $\lim_{k\to\infty} h_k = h$ we also have $\lim_{k\to\infty} f(h_k) = f(h)$. Continuity means that the emission of one output message should not depend on the reception of infinitely many input messages.

The main contribution of Kahn's paper is the *Kahn principle*, which states that the monotony and continuity of the individual Kahn processes implies the monotony and continuity of the stream function associated with the whole process network. Moreover, the stream function associated with the KPN can be computed as the least fixed point of the system of equations:

$$(Hist(co_j))_{j=1}^{m} = f((Hist(ci_j))_{j=1}^{n}) \text{ for all } f \in P$$

which can be computed iteratively. The Kahn principle thus gives the means for constructing in a compositional fashion deterministic systems.

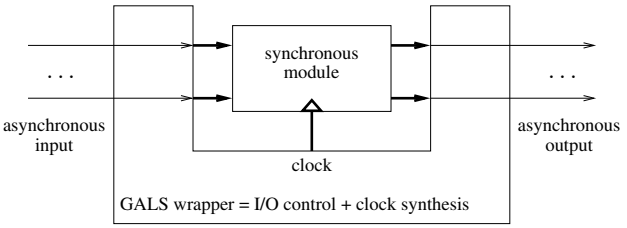


Figure 1: Desired GALS component structure. The synchronous module can be software (a reaction function) or hardware (a circuit).

2.2 The problem we address

The notions of monotony and continuity are of particular importance for us, because they are not restricted to the language defined by Kahn, nor to sequential programs. Provided we ensure the monotony and continuity of an asynchronous component, we can apply the Kahn principle.

In this paper, we address the construction of deterministic globally asynchronous systems starting from synchronous specifications. More precisely, **we focus on the problem of constructing one asynchronous process with monotonous and continuous stream function from one concurrent synchronous module**. Then, the Kahn principle can be used to compose such asynchronous processes into a deterministic system.

This paper does not deal with global correctness properties of a Kahn process network, such as the absence of overflows or deadlocks, nor with the ways of implementing and ensuring the fairness of the underlying physical computing architecture. It does not cover, either, the related problem of preserving the synchronous composition semantics in the implementation of multi-module synchronous specifications. We only focus here on the interface between the synchronous and the asynchronous parts of a single process.

2.3 Asynchronous component

We are targeting implementations having the structure depicted in Fig. 1, which are best described as *globally asynchronous, locally synchronous (GALS)*. At the core stands the synchronous module, which is driven by *an execution machine* (also called *GALS wrapper*), which allows the execution of the synchronous module in the asynchronous framework defined above. This general pattern covers a large class of implementations. For instance, the synchronous module can be a sequential reaction function (in software), or a synchronous IP (in hardware).

The execution machine performs the following functions:

- *Reaction triggering.* The successive reactions of the synchronous module are triggered, using for instance a clock generation mechanism (in synchronous hardware), or successive calls of the reaction function (in software).
- *I/O handling.* Handle the communication with both the asynchronous environment and the synchronous module, and realize the necessary transformations between the two (*e.g.* signal absence encoding, if any).

The resulting GALS component must satisfy two main *correctness conditions*:

- *Semantics preservation*. It must preserve, in some sense, the semantics of the original synchronous module.

- *Monotony and continuity*. Its stream function must satisfy the hypothesis of the Kahn principle.

We shall formally define implementation correctness in Section 2.5.1.

In addition, we make an assumption that we consider realistic for any implementation, by requiring that all resulting GALS implementations must be is *confluent*: For given, finite asynchronous inputs, any two complete executions of the implementation end up in the same state, regardless of the order in which the inputs arrive.

The form and the capabilities of the execution machine depend on the underlying hardware platform, and they must match the properties of the synchronous module to allow implementation.[1] This means that the discussion concerning the form of the execution machine cannot be done at this point. We defer it to the following sections.

2.4 Synchronous module

The various synchronous formalisms [9] are used to develop specifications that can be interpreted as *incomplete synchronous Mealy machines*. This is the model we use throughout this paper to represent synchronous modules. A module is any finite automaton whose transitions are labeled with *reactions*. An *execution (trace)* of the module is a sequence of reactions indexed by the *global clock*.

A reaction is a valuation of the *input and output signals* of the module. All signals are typed. We denote with $\mathcal{D}_S$ the domain of a signal S. Not all signals need to have a value in a reaction, to model cases where only parts of the module compute. We will say that a signal is *present* in a reaction when it has a value in $\mathcal{D}_S$. Otherwise, we say that it is *absent*. Absence is simply represented with a new value $\perp$, which is appended to all domains $\mathcal{D}_S^{\perp} = \mathcal{D}_S \cup \{\perp\}$. With this convention, a reaction is a valuation of *all* the signals S of the module in their extended domains $\mathcal{D}_S^{\perp}$. We say that two reactions r_1 and r_2 are *non-contradictory*, denoted $r_1 \bowtie r_2$, when there exists no signal S that is present, but different in the two reactions $\perp \neq r_1(S) \neq r_2(S) \neq \perp$. The *support* of a reaction r, denoted $supp(r)$, is the set of present signals. Given a set of signals X, we denote with $\mathcal{R}(X)$ the set of all reactions over X. When $r \in \mathcal{R}(X)$ and $X' \subseteq X$, then we denote with $r\mid_{X'}$ the restriction of r to X'.

To represent reactions in a compact form, we use a *set-like notation* and omit signals with value $\perp$. For instance, the reaction associating 1 to A, $\top$ to B, and $\perp$ to C is represented with $< A = 1, B = \top >$. The delimiters can be dropped if there is no confusion. On non-contradictory reactions we define the union ($\cup$) and difference ($\setminus$) operators, with their natural meanings from set theory. For instance, $< A = 1, B = \top > \cup < A = 1, C = 7 > = < A = 1, B = \top, C = 7 >$.

When representing a reaction r, we shall usually separate the valuations of the input and output signals $r = i/o$, where i is the restriction of r on input signals, and o is the restriction on output signals. All the operators defined above ($\bowtie$, $\cup$, $\setminus$, $supp()$) can be applied on components. With these conventions, the *stuttering reaction* assigning $\perp$ to all input and output signals is denoted $/$.

DEFINITION 1 (INCOMPLETE MEALY MACHINE). *A synchronous automaton is a tuple $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$, where $\mathcal{I}$ and $\mathcal{O}$ are the finite and disjoint sets of input and output signals, $\mathcal{S}$ is the set of states, and $\mathcal{T} : \mathcal{S} \times \mathcal{R}(\mathcal{I}) \longrightarrow\!\!\!\!\circ \; \mathcal{S} \times \mathcal{R}(\mathcal{O})$ is the function representing the transitions. The function $\mathcal{T}$ is partial to represent the fact that the system may not accept any input. In addition, we require that $\mathcal{I}$ is non-void.*[2]

We will write $s \xrightarrow{i/o} s'$ instead of $\mathcal{T}(s, i) = (s', o)$ to represent system transitions. Note that the functional definition of the transitions implies *determinism* (at most one transition for given state and input), a property we require for all synchronous modules throught this paper.

We denote with $Traces(\Sigma) \subseteq \mathcal{R}(\mathcal{I} \cup \mathcal{O})^{\omega}$ the set of traces of the synchronous module Σ. The determinism of Σ implies that we can also see it as a function converting sequences of input events into sequences of output events:

$$\Sigma : \; Traces(\Sigma)\mid_{\mathcal{I}} \longrightarrow Traces(\Sigma)\mid_{\mathcal{O}}$$

A Mealy machine can stutter in state s if the *stuttering transition* $s \xrightarrow{/} s$ is defined. We say that a machine is *stuttering-invariant* when there is a stuttering transition in each state. We say that a machine *has no input-less transition* if the only transitions $s \xrightarrow{i/o} s'$ with $supp(i) = \emptyset$ are stuttering transitions. In other words, no state change can be realized and no output can be produced without new inputs.

Systems that have input-less transitions, even deterministic ones, require special attention in order to produce monotonous and continuous asynchronous implementations.[3] To simplify the presentation, we shall assume that all synchronous systems of this paper have no input-less transitions.

2.5 The implementation problem

The main issue in specifying asynchronous components using synchronous specifications is the treatment of *signal absence*. In the synchronous model, the absence of a signal in a given reaction can be sensed and tested in order to make decisions. It is a special *absent* value, denoted $\perp$. In the considered asynchronous implementation model, the absence of a message on a channel cannot be sensed or tested.

When transforming the synchronous specification into a globally asynchronous implementation, the sequences of present and absent values on each signal are mapped into sequences of messages sent or received on the associated communication channels. To simplify the problem, we assume one asynchronous FIFO channel is associated with each signal of the synchronous model.

We have to define the encoding of signal values with messages on channels. When a signal S has value $v \neq \perp$ during a reaction, the most natural encoding associates one message carrying value v on the corresponding channel. The message is sent or received, depending on whether S is an output or an input signal. We assume this encoding throughout the paper.

Things are more complicated for *absent* ($\perp$) values. The most natural solution is to represent them with actual message absence

[1] For instance, if the reactions of the synchronous module are triggered periodically (as opposed to controlled by the execution machine), the module will have to be stuttering-invariant (defined in the next section), so that the synchronous module can make stuttering transitions while the execution machine waits for the inputs needed to perform computations.

[2] Our goal is the transformation of synchronous automata in *monotonous* asynchronous components. Monotony implies determinism, and input-less components, such as sensors, are deterministic only when their output values are not important (which is rarely the case in practice). We explain in Section 2.6 how our work can be extended to cover input-less components.

[3] For instance, some form of execution fairness is needed when a synchronous automaton contains a cycle formed of input-less transitions (to ensure that the cycle does not monopolize the computing power of the implementation).

```
module PREEMPT:
input A,C ; output B,D ;
abort
  await immediate A ; emit B
when immediate C do emit D end
end module
```

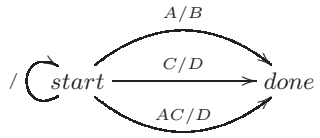


Figure 2: A small Esterel program (top), and its Mealy machine representation (bottom)

(i.e. no message at all). Unfortunately, forgetting all absence information does not allow the construction of deterministic globally asynchronous implementations for general synchronous specifications. Consider, for instance, the Esterel program of Fig. 2. The program awaits for the arrival of at least one of its two input signals. If A arrives alone, then the program terminates by emitting B. If C arrives alone or if A and C arrive at the same time, then the program terminates by emitting D.

Assume that A arrives in the $start$ state. Then, we need to know whether C is present or absent, to decide which of B or D is emitted. We will say that the program *reacts to signal absence*, because the presence or absence of C must be tested. An asynchronous implementation of PREEMPT needs absence information in order to deterministically decide which transition to trigger in state $start$.

To generate deterministic asynchronous implementations for synchronous programs such as PREEMPT, messages must be added to represent the necessary absence information. This can be done either by transmitting *absent* ($\perp$) values through messages, or by adding other synchronization messages on new or already existent communication channels.

In this paper, our objective is to characterize the class of synchronous specifications that can be transformed into monotonous and continuous asynchronous implementations without adding such new messages to encode absence. Our characterization has important consequences. In particular, it establishes the theoretical limits of the two-phase implementation process *à la* Signal/Polychrony [8], where all absence encoding problems are dealt with *inside the synchronous model*, thus facilitating the analysis of large specifications using existing synchronous tools and techniques:

Step 1: Signal absence encoding. Transform the synchronous specification into one where reaction to signal absence is not needed.[4] This is done by either (1) deciding which $\perp$ values are relevant and must be transmitted, and represent them with a new value $\perp^*$, or (2) adding new signals and messages to the specification.

Step 2: Implementation synthesis. Give a deterministic asynchronous implementation to the transformed synchronous specification. This implementation follows the natural encoding defined above: no message for the remaining *absent* $\perp$ values, and one message for each other signal value.

[4]The transformation can be automatic, as Polychrony does for simulation purposes, or manual, when building an implementation that must be finely tuned to match the underlying execution platform.

2.5.1 Formal correctness criterion

Assume that $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$ is a synchronous module and that $[\Sigma]$ is his GALS implementation. As stated above, there is a 1-to-1 correspondence between the signals of Σ and the channels of $[\Sigma]$. By an abuse of notations, we identify the set of channels C of $[\Sigma]$ with $\mathcal{I} \cup \mathcal{O}$.

The encoding of signal absence with actual absence of messages is represented using the *desynchronization operator* $\delta()$, which converts synchronous traces (sequences of reactions) into asynchronous histories by forgetting $\perp$ values. On individual signals/channels:

$$\delta() : {\mathcal{D}_S^{\perp}}^{\omega} \rightarrow {\mathcal{D}_S}^{\omega} \qquad \delta(v) = \begin{cases} \epsilon, \text{ if } v = \perp \text{ or } v = \epsilon \\ v, \text{ if } v \in \mathcal{D}_S \\ \delta(u)\delta(w), \text{ if } v = uw \end{cases}$$

For a set of signals/channels X:

$$\delta() : \mathcal{R}(X)^{\omega} \rightarrow \prod_{S \in X} {\mathcal{D}_S}^{\omega} \qquad \delta(t)(S) = \delta(t \mid_{\{S\}})$$

We have now all the formal elements needed to define the desired sematics preservation criterion the GALS implementation must satisfy. Assume t is a synchronous trace of Σ. When feeding the GALS implementation $[\Sigma]$ with $\delta(t \mid_{\mathcal{I}})$, which contains all the inputs values different from $\perp$, we expect as output $\delta(t \mid_{\mathcal{O}})$:

$$[\Sigma] : \delta(t \mid_{\mathcal{I}}) \mapsto \delta(t \mid_{\mathcal{O}}) \tag{1}$$

Generally, this mapping does not define, by itself, a monotonous and continuous stream function. Two problems arise:

- The mapping is usually incomplete, as the inputs of synchronous traces do not cover all possible input histories.[5] We must extend it to a full stream function.
- The mapping cannot always be extended to a monotonous and continuous function. For instance, this is impossible for the synchronous module in Fig. 2.

We will say that the GALS implementation $[\Sigma]$ is correct when it is confluent and has a monotonous and continuous stream function ensuring the mapping of Equation 1.

This criterion specializes and enriches the semantics-preservation criterion originally defined by Benveniste, Caillaud and LeGuernic [3] by taking into account the properties of our implementation framework.

We shall see in the following sections how the execution machine further limits the class of programs that can be implemented, beyond the limits imposed by the previous criterion.

2.6 Related work

Our work is closely related to, and can be seen as a generalization over, the multiple variants of *endochrony*[3, 8, 14, 13]. Indeed, we determine here necessary and sufficient conditions for deterministic and confluent asynchronous implementation of a synchronous specification, where the various endochrony variants offer only sufficient conditions. The fact that our formalization covers only deterministic systems, whereas endochrony also also allows some non-determinism, is due to a presentation choice. As explained in [14], the Kahn principle can be generalized to ensure a form of *predictability*[6] instead of determinism. Our results can be

[5]For instance, an adder needs both inputs at each reaction, so the mapping is defined only for input histories with the same number of values on each channel.

[6]Non-deterministic internal choices of a process are possible, as long as they are published through the outputs, allowing the environment to change its behavior accordingly.

easily generalized following the same scheme. This generalization should also allow the modelling of sensors (which is impossible in the current KPN-based setting, because a source is usually non-deterministic).

It must be noted, however, that we pursue slightly different goals than the endochrony-based approaches. More precisely, we do not address here the preservation of synchronous *composition* semantics in a *distributed* asynchronous setting. This is why we do not define a second property (like *(weak) isochrony* [3, 14]) covering composition, and instead focus on the transformation of the synchronous specification of a single component into an asynchronous implementation.

We determine that the various formulations of (weak) endochrony aim at expressing a more fundamental property of a concurrent synchronous specification: **the fact that it does not react to signal absence**.

Work on endochrony also justifies our approach from a practical point of view. First, the two-step implementation process we advocate for generalizes the process of the Polychrony [8, 1] environment developed around the Signal language and the endochrony variant of Benveniste, Caillaud, and Le Guernic [3]. Second, we establish limits that must be respected by any globally asynchronous implementation.

Technical comparisons with the endochronous systems of Benveniste, Caillaud, and Le Guernic [3] and with the (microstep) weakly endochronous systems of Potop, Caillaud, and Benveniste [14, 13], will be respectively provided in Sections 4.1 and 5.2.1.

The endochrony variant of LeGuernic, Talpin, and Le Lann [8] has a more marked difference with respect to our model. It only requires that the system produces correct output for specified sets of inputs, which leads to composition problems and more complex analysis techniques. [7] By comparison, our GALS components are deterministic for any input (like Kahn processes do).

The *latency-insensitive systems* of Carloni, McMillan, and Sangiovanni-Vincentelli, the Lustre language [10], and the AAA/SynDEx methodology [7] take a very simple solution to the absence encoding problem, by prohibiting the absence of interface signals. This means that the programmer must perform the absence encoding step, and that all concurrency is lost in the system (the approach is not very efficient). The *generalized latency-insensitive systems* of Singh and Theobald [15] try to relax these constraints.

Our results do not cover the distributed implementation of reactive systems, as do Caspi, Girault, and Pilaud [6], nor have the same global approach. We only deal with the construction of one deterministic asynchronous component from one synchronous specification.

From another perspective, the synchronous systems, as defined in distributed computing [2] correspond in our case to (concurrent) synchronous specifications without reaction to signal absence.

Our work has different goals from Boussinot and de Simone's work on *instantaneous reaction to signal absence* [5]. There, the issue is that of determining signal absence while avoiding causality problems.

3. SIGNAL ABSENCE IN VARIOUS LANGUAGES

Programs written in the three main synchronous languages are easily represented with our Mealy machines.

For the Signal language, we consider its trace semantics, as defined in [1]. A reaction of the program is a partial assignment of the signals that satisfies the constraints represented by the statements of the program. When a reaction does not assign a signal, we say that the signal is absent. This absence representation is naturally mapped to our absence encoding which uses explicit $\perp$ values. We shall denote with $\top$ the unique present value of the signals of type `event`. A Signal program has no explicit global clock, so that all Signal programs are stuttering-invariant (the stuttering transition is defined in all states), but all Signal programs are not deterministic. For the scope of this paper, we shall only consider deterministic Signal programs.[8] Stuttering-invariance and determinism imply that the programs we consider have no input-less transition.

Lustre and the specification formalism of the SynDEx software can be seen as sub-sets of Signal, the main differences being that the global clock is specified, and that no interface signal is ever absent (by consequence, no input-less transition exists). Thus, we can use the same encoding as for Signal.

In Esterel, every signal has a *status* of *present* (true) or *absent* (false).[9] Valued signals also carry a value, that should be read only during reactions where the signal is present (*status=true*). The mapping from the Esterel absence encoding to ours is again natural: a signal which is present is represented by its value (if the signal is valued), or by $\top$ (if the signal is not valued). A signal that is absent has value $\perp$.

The Esterel language defines a global clock. Time flow, and therefore the reactions where a signal can be absent, is determined by the successive occurrences of the implicit `TICK` signal. In particular, no other signal can be present if `TICK` is absent.

Esterel programs can have input-less transitions. The following program can produce `O` without reading a single input (`TICK` is not an input signal):

```
module TIMEFLOW:
output O;
loop
  every 2 TICK do emit O end
end
end module
```

However, meaningful classes of Esterel programs exist without input-less transitions. Such a class is the syntactical sub-set of Esterel containing no `pause` or `suspend` statements and no reference to `TICK`, and where all preemption triggers are reduced to one signal.

To represent the behavior of `TIMEFLOW` with a deterministic Signal program (without input-less transitions), the `TICK` signal must be explicitly represented in the input interface of the program, for instance:

```
1 process TIMEFLOW =
2 (? event TICK ; ! event O ;)
3 (| State ^= TICK
4  | State := preState $init (-1)
5  | preState := (State+1) modulo 2
6  | O := when State=1
7  |) where integer State,preState ; end ;
```

Line 3 specifies that the state is read and updated whenever `TICK` is present.

Representing the behavior of `TIMEFLOW` in Lustre leads to different problems. Like in Esterel, Lustre programs define a notion of global clock. However, the use of absence is constrained. More precisely, all the inputs and outputs of a Lustre program (node)

[7] This is similar to hardware-like *don't care*-based approaches.

[8] Only such programs can be given deterministic asynchronous implementations without signal absence encoding.

[9] We consider here only correct programs, and ignore all causality issues.

need to be present at all instants where the node is executed. This means that the previous example cannot be encoded while sticking to the absence encoding defined above. The only solution is to explicitly encode absence using "present" signal values. One typical solution is to use a Boolean signal with the same encoding as the one used for signal statuses in the compilation of Esterel (present=true/absent=false):

```
node TIMEFLOW() returns (O:boolean);
var state : integer ;
let
  state = (-1) -> (pre(state)+1) mod 2 ;
  O = state==1 ;
tel
```

Not having a dedicated signal absence representation for interface signals means that a Lustre program uses one message per absent value. The specification formalism of the SynDEx software has roughly the same constraints as Lustre. The Scade formalism – the graphical counterpart of Lustre – relaxes this rule, but still does not allow the direct representation of the previous example.

4. REACTION TO SIGNAL ABSENCE

In this section, we formally define reaction to signal absence and we explain how synchronous specifications without reaction to signal absence can be given deterministic asynchronous implementations.

We say that a system reacts to signal absence when the choice between two transitions in a state is based on the choice over the present/absent value of a signal. Formally, it is simpler to define the dual property:

DEFINITION 2 (NO REACTION TO SIGNAL ABSENCE (NRSA)). *Let $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$ be a synchronous Mealy machine. We say that Σ does not react to signal absence if for every state s and every two non-stuttering transitions $s \xrightarrow{r_k = i_k/o_k} s_k$, $r_k \neq /$, $k = 1, 2$, we have:*

$$r_1 \neq r_2 \Rightarrow \exists S \in \mathcal{I} : \bot \neq i_1(S) \neq i_2(S) \neq \bot$$

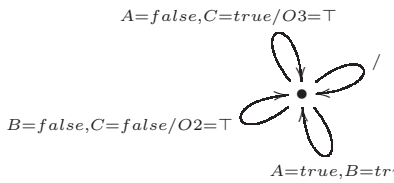
In other words, we can decide which transition to do by testing the value (and not the presence/absence) of an input. This choice can be implemented as a deterministic choice in our asynchronous framework.

4.1 Asynchronous implementation issues

The NRSA criterion preserves the spirit of endochrony, as defined in [3], but strictly generalizes it. The following example shows the difference between NRSA and endochrony:

```
process NOABSENCE1 =
(? boolean A, B, C ;
 ! event O1, O2, O3;)
(| O1 ^= when A=true  ^= when B=true
 | O2 ^= when B=false ^= when C=false
 | O3 ^= when A=false ^= when C=true
 |)
```

The corresponding automaton in our model is the following:



Choosing between the three non-stuttering transitions can be done without signal absence information, so deterministic implementation is possible in an asynchronous environment.

However, the program is not endochronous in the sense of [3]. In an endochronous program, the signals can be organized in a decision tree (called *clock tree*). Input reading starts with the signals at the top level of the tree, which are present in each reaction. Depending on their values, some of their direct children are read, and the process continues recursively from each present signal to its present children signals. A signal is present in the current reaction *iff* its clock tree node has been traversed. *Blocking reads* can be used to produce a *fully deterministic* top-down input reading process. This form of endochrony stands at the basis of the Signal compiler [1].

In our example, the signals cannot be organized in a tree determining which signals are present in an incremental fashion. Blocking reads can no longer be used. Instead, each input FIFO must deliver messages as they arrive. Once an input message m arrives on the FIFO head f_S corresponding to signal $S \in \mathcal{I}$, f_S will accept no more messages until a reaction consumes the value $v(m)$ of m (*i.e.* until a reaction i/o is executed so that $i(S) = v(m)$). A fireable reaction i/o can be triggered as soon as its present input values are available as messages on the input FIFO heads corresponding to the present signals of i. Once this condition is met, the actual transition can be triggered in a variety of ways, without affecting the functionality and determinism of the implementation: by some external clock (periodic or not), when enough input is available to trigger a non-stuttering reaction, etc.

4.1.1 Incremental ASAP input reading

One possible asynchronous execution of the previous example is given in Fig. 3. It corresponds to a GALS implementation where reactions are triggered by an external clock. The input FIFO associated with signal C is the first to deliver a value (false). Then, new values arrive for B and A. When a reaction is triggered by the external clock, the only non-stuttering fireable reaction is $A = true, B = true/O1 = \top$. This reaction is performed, O1 is emitted, the first messages on FIFOs A and B are consumed (new messages can arrive), but the message on FIFO C remains unconsumed. After a new value arrives for A, a new reaction is triggered by the external clock. Given the available inputs, the only fireable transition is $\bot$, which changes nothing. The third reaction is $B = false, C = false/O2 = \top$.

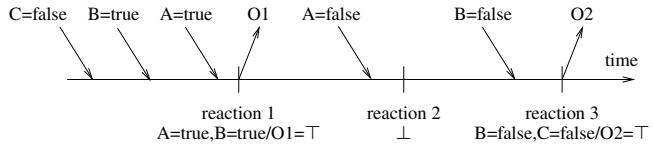


Figure 3: Incremental ASAP asynchronous execution of NOABSENCE1

Note that, in our example, we always perform a reaction as soon as its inputs are available at clock activation time (never delaying execution). This choice is natural, as it minimizes the number of clock cycles needed to complete a computation. We shall say that reactions are executed *as soon as possible (ASAP)*.

Also note that the NRSA property allows an incremental reading of the inputs needed to trigger a reaction r. As soon as the system enters a state where $r = i/o$ is fireable, we can start a process $Wait_i$ that waits for the values of i to arrive on the input FIFOs. Once the inputs are assembled, r can be executed ASAP. The input

reading process $Wait_i$ is killed if some input FIFO f_S with $S \in supp(i)$ brings a message m with $i(S) \neq v(m)$.

In Fig. 3, for instance, the input reading processes $Wait_{A=true,B=true}$, $Wait_{B=false,C=false}$, and $Wait_{A=false,C=true}$ start when execution starts. We focus on $Wait_{B=false,C=false}$. It assembles $C = false$, but $B = true$ arrives and $Wait_{B=false,C=false}$ is killed. However, the reaction $B = false, C = false/O2 = \top$ is again fireable after the execution of the first reaction. Therefore, $Wait_{B=false,C=false}$ is restarted. It assembles $C = false$, which is still unconsumed. Finally, $B = false$ arrives and the $Wait_{B=false,C=false}$ completes its execution by triggering the associated reaction at the third activation of the clock (as soon as possible). When ASAP execution is combined with this incremental input reading mechanism, we shall say that we have an *Incremental ASAP* input reading (and reaction triggering) policy.

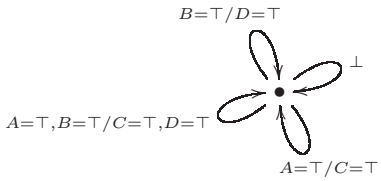
Incremental ASAP input reading is more complex than endochronous input reading but allows the deterministic asynchronous implementation of more synchronous systems. The basic Incremental ASAP technique described here can be optimized, *e.g.* by using blocking reads (like in the endochronous approach) every time this is possible. But we shall not cover optimization aspects here.

5. EXTENSION TO CONCURRENT SYSTEMS

The NRSA property ensures that for given input messages, the program can choose deterministically the non-stuttering reaction to trigger. However, this strong form of determinism is not always necessary to ensure the I/O determinism of an asynchronous implementation. Consider the following Signal program:

```
process WE1 =
(? event A, B ; ! event C, D ;)
(| C := A | D := B |)
```

The corresponding automaton in our model is the following:



The automaton does not satisfy the NRSA property, because absence is needed to choose between $A = \top/C = \top, B = \top/D = \top$, and $A = \top, B = \top/C = \top, D = \top$. However, the program can be asynchronously implemented, without added signalling, if we use an Incremental ASAP input reading and reaction triggering policy. Indeed, as soon as A is received, the reaction $A = \top/C = \top$ can be executed, whether B has been received or not.

Intuitively, reactions $A = \top/C = \top$ and $B = \top/D = \top$ are *independent*. The interleaving between incoming messages on the A and B channels, and the associated interleaving of reactions do not change the asynchronous I/O behavior of WE1. The same is true for the corresponding Esterel program:

```
module WE1: input A,B ; output C,D ;
[
  every immediate A do emit C end
||
  every immediate B do emit D end
]
end module
```

More generally, the *weak endochrony* property introduced by Potop, Caillaud, and Benveniste [14] ensures that an Incremental ASAP input reading policy produces deterministic asynchronous implementations.

While weak endochrony is a sufficient condition, we determine here the exact class of synchronous programs (automata) that produce deterministic asynchronous implementations when an Incremental ASAP policy is used.

5.1 Concurrent Incremental ASAP

The first step in this direction is to determine that the Incremental ASAP policy is compatible with concurrent systems such as WE1.

In systems with the NRSA property, at most one non-stuttering reaction r is executed during a clock activation because at most one input gathering process can complete between two clock activations. This is no longer the case when concurrent reactions are accepted. For instance, assume that both inputs A and B arrive before the first clock activation of example WE1. Three input gathering processes are started (one for each non-stuttering fireable reaction), and all three are completed before the first clock activation. In the end, to avoid ambiguity, only one should be executed. In our case, the combined reaction $A = \top, B = \top/C = \top, D = \top$.

To obtain this behavior, we extend Incremental NRSA with two new rules allowing the handling of non-contradictory concurrent reactions. Assume that two input reading processes are started for reactions i_1/o_1 and i_2/o_2, with $i_1 \bowtie i_2$ and $supp(i_1) \cap supp(i_2) \neq \emptyset$. The rules are the following:

SR1 (competition for resources) If $Wait_{i_1}$ is completed and $Wait_{i_2}$ is not, then i_1/o_1 is executed and $Wait_{i_2}$ is killed.

SR2 (the bigger transition wins) If both $Wait_{i_1}$ and $Wait_{i_2}$ are completed and $i_1 \subset i_2$, then i_2/o_2 is executed and $Wait_{i_1}$ is killed.

We shall call this extended input reading policy *Concurrent Incremental ASAP*.

Note that the correctness of an Incremental ASAP policy (concurrent or not) relies on two fundamental properties:

FP1 The scheduling rules leave at most one transition executable at each activation of the clock.

FP2 A reaction i/o is still fireable when $Wait_i$ is completed. This means that the transitions realized from the moment $Wait_i$ is started and until it completes without being killed leave the reaction fireable.

These properties are implied by the NRSA property, and need to be preserved by its extension to concurrent systems.

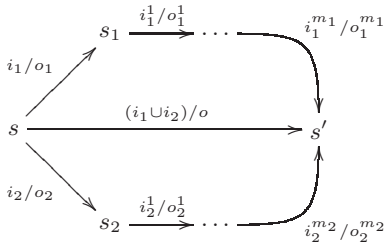
5.2 Concurrent NRSA

Consider a synchronous specification $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$ with the property that its implementation $[\Sigma]$ using a Concurrent Incremental ASAP input reading policy is monotonous and deterministic.

Assume that Σ is in state s and that $s \xrightarrow{r_k = i_k/o_k} s_k$, $k = 1, 2$ such that r_1 and r_2 are not stuttering transitions and $i_1 \bowtie i_2$. Assume that all the inputs of $i_1 \cup i_2$ arrive through messages before a new activation of the clock. Then, $Wait_{i_1}$ and $Wait_{i_2}$ are both completed, but to comply with property FP1 only one reaction must be executed. Rule SR1 cannot be used to make this choice, meaning that we have to use rule SR2. This means that the reaction i/o that is executed must satisfy $i_1 \subseteq i$ and $i_2 \subseteq i$. At the same time, given the available input, we also need $i \subseteq i_1 \cup i_2$. Therefore, $i = i_1 \cup i_2$. Also, from the I/O monotony of the asynchronous implementation,

and from the fact that Σ has no input-less transitions, we have $o_1 \subseteq o$ and $o_2 \subseteq o$, and therefore $o_1 \cup o_2 \subseteq o$. In other words, there exists $s \xrightarrow{(i_1 \cup i_2)/o} s'$ with $o_i \subseteq o, i = 1, 2$.

From the confluence property required for the asynchronous implementations in Section 2.3, and from the restriction to systems that have no input-less transitions, we can deduce that there exist the reactions i_j^k/o_j^k with $j \in \{1, 2\}$ and $0 \leq k \leq m_j$ such that:



$(cnrsa)$

where the following equalities hold, their terms being defined, and defining partitions (in terms of support) of their left terms:

$$i_2 \setminus i_1 = \bigcup_{j=1}^{m_1} i_1^j \qquad i_1 \setminus i_2 = \bigcup_{j=1}^{m_2} i_2^j \tag{2}$$

$$o = o_1 \cup \bigcup_{j=1}^{m_1} o_1^j = o_2 \cup \bigcup_{j=1}^{m_2} o_2^j \tag{3}$$

This is the property defining concurrent systems without reaction to signal absence.

DEFINITION 3 (CONCURRENT NRSA). *Given a synchronous specification $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$, we shall say that it satisfies the* no reaction to signal absence property for concurrent systems *(Concurrent NRSA) if for every two transitions $s \xrightarrow{i_k/o_k} s_k$, $k = 1, 2$ such that $i_1 \bowtie i_2$, there exist $s' \in S$ and o valuation of the output signals such that $o_1 \cup o_2 \subseteq o$, as well as the reactions i_j^k/o_j^k with $j \in \{1, 2\}$ and $0 \leq k \leq m_j$ satisfying equations (cnrsa), (2), and (3).*

The following theorem is our main result. It proves that the Concurrent NRSA property indeed characterizes synchronous specifications that give deterministic asynchronous implementations when a Concurrent Incremental ASAP input reading policy is used.

THEOREM 1 (CHARACTERIZATION). *Let $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$ be a synchronous specification with no input-less transitions. Then, the Concurrent Incremental ASAP asynchronous implementation of Σ is confluent, monotonous and deterministic if and only if Σ has the Concurrent NRSA property.*

Proof: $\Rightarrow$. Already done, in a constructive fashion, in this section. $\Leftarrow$. Given the form of our implementations, proving confluence and monotony is sufficient (continuity is implied, as no output message needs to wait for an infinity of input messages).

Consider now a finite input history (the result for infinite input histories is a consequence of continuity). Also consider two arrival orders of these incoming inputs with respect to the clock triggering instants, and consider the associated maximal Concurrent Incremental ASAP executions without final stuttering transitions. Given that every transition (stuttering ones excepted) consumes at least one input, the two executions are finite.

To prove confluence, we need to prove that the two executions read the same input messages, produce the same output messages, and end up in the same state.

Note that confluence implies monotony. Indeed, consider two finite input histories $\chi \preceq \chi'$. Then, the confluence result allows us to consider for the input χ' a maximal execution that is a completion of a maximal execution for χ (by assuming that the inputs of χ' that are not in χ arrive only after the maximal execution for χ is completed). This proves monotony.

The proof of confluence (all that remains to be done) is based on Lemma 2. This lemma, when applied to maximal traces with the same asynchronous input gives us the needed confluence result. **End proof.**

LEMMA 2 (CONFLUENCE). *Consider the traces $t_i \in Traces(\Sigma)$, $i = 1, 2$ such that for all $S \in \mathcal{I}$ we have $\delta(t_1)(S) \preceq \delta(t_2)(S)$ or $\delta(t_2)(S) \preceq \delta(t_1)(S)$. Then, t_i can be extended to $t_i' \in Traces(\Sigma)$, $i = 1, 2$, such that $\delta(t_1') = \delta(t_2')$, $\delta(t_1' \mid_{\mathcal{I}}) = \delta(t_1 \mid_{\mathcal{I}}) \vee \delta(t_2 \mid_{\mathcal{I}})$. Moreover, the execution of both t_1' and t_2' ends in the same state.*

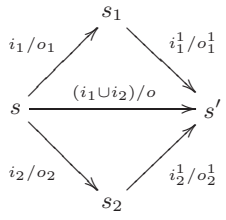
In other words, if two traces have inputs that are asynchronously compatible, then they can be seen as partial executions of $[\Sigma]$ for the input $\delta(t_1 \mid_{\mathcal{I}}) \vee \delta(t_2 \mid_{\mathcal{I}})$ (on each input channel S, the maximum of $\delta(t_1)(S)$ and $\delta(t_2)(S)$). Then, each trace can be completed to one that reads the whole input, and the destination state is the same.

For space reasons, we do not give here the proof of the lemma. It is done by induction over the lengths of the two traces.

5.2.1 Relation with weak endochrony

Consider a synchronous system Σ satisfying the Concurrent NRSA property and two non-contradictory transitions $s \xrightarrow{i_k/o_k} s_k$, $k = 1, 2$. Then, there exists $s \xrightarrow{i_1 \cup i_2/o} s'$ with $o_1 \cup o_2 \subseteq o$. Assume the inputs of i_1 arrive before the first activation of the clock, and that the inputs of $i_2 \setminus i_1$ arrive before the second activation of the clock. From the determinism of the asynchronous implementation, we know that all the inputs of $i_1 \cup i_2$ will be read, and all the outputs of o produced. However, we have no guarantee on the number of clock activations needed to consume the inputs of $i_2 \setminus i_1$, and to produce the outputs of $o \setminus o_1$.

The most natural way to limit the number of clock activations needed to ensure confluence is to require that the outputs of o are all produced by the end of the reaction where the last inputs of $i_1 \cup i_2$ are consumed. Formally, this strengthens axiom (*cnrsa*) into:



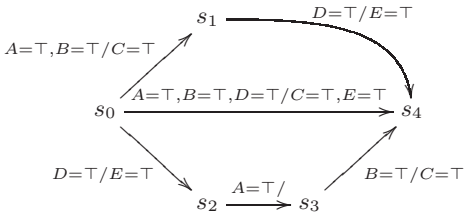
$(cnrsa - fast)$

with $i_1^1 = i_2 \setminus i_1$, $o_1^1 = o \setminus o_1$, $i_2^1 = i_1 \setminus i_2$, and $o_2^1 = o \setminus o_2$.

This property is just a sufficient condition ensuring monotonous and deterministic Concurrent Incremental ASAP execution, but it allows more efficient implementations. Note that it can be seen as the macrostep version of microstep weak endochrony [13]. Properties *(cnrsa)* and *(cnrsa-fast)* are not compositional, but this is not an issue here, because we are only considering one implementation level (the interface with the runtime) and no hierarchy.

An example of synchronous system satisfying the Concurrent NRSA property, but which cannot be represented with a microste

weakly endochronous automaton is:



An even stricter restriction consists in requiring that rule SR1 is never applied, which greatly simplifies the structure of the asynchronous implementation. In this case, we are lead to a decomposition of all executions into atomic reactions. The result is macrostep weak endochrony [14].

Note how fine tuning efficiency and the capabilities of the execution machines leads to various sufficient conditions for deterministic desynchronization.

6. EXAMPLES

All Lustre programs are endochronous in the sense of Benveniste [3], and therefore satisfy property NRSA.

But it is more interesting to explain which common properties of a synchronous program mean that it does not satisfy the NRSA or Concurrent NRSA properties. We give here intuitive examples where signal absence information is necessary for the deterministic asynchronous implementation of a synchronous system.

6.1 Preemption (a simple example in Esterel, Signal, and Lustre)

Consider the Esterel example `PREEMPT`, of Fig. 2. We recall the program body:

```
abort
  await immediate A ; emit B
when immediate C do emit D end
```

We explained in section Section 2.5 that the program does not have the NRSA property. If we use no message to encode signal absence, the resulting asynchronous implementation is non-deterministic: For given input (one message on channel `A`, and one message on channel `C`) 2 different outputs can be obtained.

A simple Signal language counterpart makes reaction to signal absence even more obvious, under the form of "`not CE`", used in the lines 8 and 11.

```
1  process PREEMPT =
2  (? event A,C,TICK ; ! event B,D ;)
3  (| state ^= AE ^= CE ^= TICK
4   | A ^+ C ^< TICK
5   | AE := (true when A) default false
6   | CE := (true when C) default false
7   | state :=
8      (state and not AE and not CE)
9      $init true
10  | B :=
11     when (state=true and AE and not CE)
12  | D := when (state=true and CE)
13  |) where boolean state, AE, CE; end
```

By comparison, encoding the previous example into Lustre (or SynDEx) requires the programmer to manually encode presence and absence with non-absent values of Boolean signals. No signal absence subsists on the program interface.

```
node PREEMPT(A,C:boolean)
returns (B,D:boolean);
var active:boolean ;
let
  active = true ->
    pre(active and not A and not C) ;
  B = state and A and not C ;
  D = state and C
tel
```

6.2 Signal loss

Programs written in Esterel and Signal can lose incoming signals. By losing signals we mean that signal valuations can be left unread (and thus discarded) without influencing the behavior of the system in any way. This is generally not acceptable when we want to achieve determinism in the chosen asynchronous framework, because we don't know the number of messages to read. We start with a simple Esterel example:

```
module LOSS1 :
input A, B ; output C,D ;
[
  await immediate A ; emit C
||
  await immediate B ; emit D
]
end module
```

When A and B arrive simultaneously, the program instantly emits C and D and terminates. Assume now that A arrives first, and that B arrives in a subsequent instant. After the reception of A and before the reception of B, the first branch is terminated, so that the program does not explicitly use incoming A signals. However, such signals can arrive (one per reaction, at most), and are lost.

By consequence, `LOSS1` does not have the NRSA property. In instants between the first occurrence of A and the first occurrence of B, the program can choose between executing $TICK = \top/$ or $TICK = \top, A = \top/$ or $TICK = \top, B = \top/D = \top$ or $TICK = \top, A = \top, B = \top/D = \top$.

The behavior of the previous Esterel example can be modelled in Signal as follows.

```
1  process LOSS1 =
2  (? event A,B,TICK ; ! event C,D ;)
3  (| A ^< TICK | B ^< TICK
4   | AE ^= BE ^= stateA ^= stateB ^= TICK
5   | AE := (true when A) default false
6   | BE := (true when B) default false
7   | stateA := stateAnxt $ init true
8   | stateAnxt := stateA and not AE
9   | stateB := stateBnxt $ init true
10  | stateBnxt := stateB and not BE
11  | C := when stateA and AE
12  | D := when stateB and BE
13  |) where
14      boolean AE,BE,stateA,stateB,
15              stateAnxt,stateBnxt ;
16    end ;
```

The way signals are lost is more obvious here. The lines 5 and 6 show how inputs are read at each reaction. At the same time, the input data is only used when `stateA`, respectively `stateB` are true.

It is important to note that all useful Esterel programs lose messages. More precisely, the only programs that do not lose inputs

are those that read all their inputs at all instants. This is due to the fact that Esterel programs do not constrain their environment, or do it in elementary ways, whereas a Signal program specifies both the system and its environment. To allow the use of Esterel for the specification of systems that do not react to signal absence, we need to constrain the environment, using a constraint language such as Signal, so that signals do not arrive when they are not awaited.

The previous Signal program, which can lose signals, can be "fixed" by requiring inputs to come only in instants where they are awaited, for instance by changing line 3 as follows.

```
(| A ^< TICK | B ^< TICK
 | A ^< when stateA=true
 | B ^< when stateB=true
```

One could imagine combining Esterel programs with Signal environment constraints, to obtain the same effect.

6.3 Signal merging and splitting

A special form of signal loss occurs when two or more statements simultaneously emit or read a signal, while reading can also be done separately:

The simplest case is that of emission, illustrated by the following Esterel program:

```
module LOSS2 : input A, B ; output C ;
[
  await immediate A ; emit C
||
  await immediate B ; emit C
]
end module
```

Depending on the arrival of A and B, the asynchronous implementation of the program can produce one or two messages on C.

The following example can read two messages for signal E (in two different reactions), or just one (when A, B, and E arrive simultaneously), or none (when no A, nor B arrive):

```
module LOSS3 :
input A, B, E ; output C, D ;
[
  await immediate [A and E] ; emit C
||
  await immediate [B and E] ; emit D
]
end module
```

7. CONCLUSION

We have introduced a simple formal definition of reaction to signal absence. We have defined the execution machine that allows the deterministic execution of synchronous programs with no reaction to signal absence (NRSA) in an asynchronous environment. We have determined a formal criterion characterizing the class of concurrent programs that are deterministic when run using this execution machine. The Concurrent NRSA criterion generalizes various notions of (weak) endochrony and establishes theoretical and practical limits for deterministic desynchronization. Intuitive examples have been used to illustrate the various concepts.

Future work will concentrate on practical application of these results to the optimization of the communication mechanisms of GALS implementations generated by systems such as SynDEx. We will also develop analysis and synthesis techniques for the deterministic asynchronous implementation of programs written in common synchronous languages.

Acknowledgements. The authors acknowledge the constructive and consistent remarks of our (anonymous) EMSOFT reviewers, which lead to important presentation improvements.

8. REFERENCES

[1] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings PLDI'95*, La Jolla, CA, USA, June 1995.

[2] H. Attiya. *Distributed Computing*. McGraw-Hill Publishing Company, 1998.

[3] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125 – 171, 2000.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan. 2003.

[5] F. Boussinot and R. de Simone. The SL synchronous language. Research Report RR-2510, INRIA, Sophia Antipolis, France, March 1995. http://www.inria.fr/rrrt/rr-2510.html.

[6] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Proceedings PDCS'94*, Las Vegas, USA, October 1994.

[7] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

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

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer academic Publishers, 1993.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North Holland, 1974.

[12] E. Lee and T. Park. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–799, 1995.

[13] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 78(1):131–159, 2007.

[14] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, March 2006.

[15] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proceedings DATE'04*, Paris, France, 2004.