Necessary and Sufficient Conditions for Deterministic Desynchronization

Dumitru Potop-Butucaru, Robert De Simone, Yves Sorel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Necessary and Sufficient Conditions for Deterministic Desynchronization

Dumitru Potop-Butucaru — Robert de Simone — Yves Sorel

N° 6152

March 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6152--FR+ENG

Necessary and Sufficient Conditions for Deterministic Desynchronization

Dumitru Potop-Butucaru , Robert de Simone , Yves Sorel

Thème COM — Systèmes communicants
Projet AOSTE

Rapport de recherche n° 6152 — March 2007 — 21 pages

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 (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" for correct desynchronization.

Key-words: Synchronous, Asynchronous, GALS, Desynchronization, Implementation, Execution machine, Reaction to signal absence

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Conditions nécessaires et suffisantes pour une désynchronisation déterministe

Résumé : Les formalismes réactives synchrones utilisent une notion d'horloge globale pour préciser l'ordonnancement d'opérations concurrentes. Cela mène à une définition non-ambigue de l'absence d'un signal, qui peut etre utilisée ensuite pour prendre des décisions au cours de l'exécution.

Quand à une spécification synchrone on donne une implantation asynchrone (potentiellement répartie), l'absence des signaux doit être signalée explicitement, sauf pour les cas où les comportements sont prouvés indépendents et asynchrones (une propriété introduite précédemment sous le nom d'"endochronie"). Dans cet article, nous caracterisons la classe de systèmes dont l'implantation asynchrone sans encodage de l'absence est déterministe.

Mots-clés : Synchrone, Asynchrone, GALS, Desynchronisation, Implantation, Machine d'exécution, Reaction à l'absence d'un signal

1 Introduction

Synchronous reactive formalisms [11, 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 on 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 extend. Thus a natural question is: *when can one dispose of such "absent signal" communications?*

Sufficient conditions, known as *(weak) endochrony* [3, 10, 19, 18], 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 [13], 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 discuss the structure of the execution machines needed 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. 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 Asynchronous components

Our goal is to facilitate the construction of globally asynchronous systems from synchronous specifications. By globally asynchronous we understand both (1) fully asynchronous systems and components and (2) globally asynchronous, locally synchronous (GALS) systems

where each component consists of a synchronous core driven by a wrapper in the globally asynchronous environment.

More specifically, the goal is to allow the construction of asynchronous components that fit inside a model of computation close to that of the Kahn Process Networks (KPN) [13]. Such components communicate through *message passing* along *asynchronous lossless communication lines (FIFO channels)*. Incoming messages remain on the FIFOs until they are read. No other communication or synchronization mechanism exists. No logical or physical time can be used to make decisions and trigger computations (only the arrival of messages, and their values).

To simplify the presentation of this paper, we require that the behavior of an asynchronous component is *monotonous* and *deterministic* as an I/O stream function, in the sense of Kahn [13].[1] We also require that the asynchronous component is *confluent*: from a given state and for given inputs, the implementation ends up in the same internal state.

The components must remain deterministic when given arbitrary inputs. This is different from approaches based on *don't cares*, which assume that the environment never produces incorrect input sequences, and provide no means of rejecting them. By comparison, in the endochrony variant of LeGuernic, Talpin, and Le Lann [10], the system must only be deterministic and produce correct output for "correct" sets of inputs, which leads to composition problems and more complex analysis techniques.

2.2 Synchronous specifications

The various synchronous formalisms 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 our synchronous specifications. A specification is therefore any finite automaton whose transitions are labeled with *reactions*. An *execution (trace)* of the specification is a sequence of reactions indexed by the *global clock*.

A reaction is a valuation of the input and output *signals* of the specification. 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 specification 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 $\bot$, which is appended to all domains $\mathcal{D}_S^\bot = \mathcal{D}_S \cup \{\bot\}$. With this convention, a reaction is a valuation of *all* the signals S of the specification in their extended domains $\mathcal{D}_S^\bot$. 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 $\bot \neq r_1(S) \neq r_2(S) \neq \bot$. The *support* of a reaction r, denoted $supp(r)$, is the set of present signals. Given a set of signals X, we denote with $\mathcal{V}_X$ the set of all reactions over X.

To represent reactions, we use a *set-like notation* and omit signals with value $\bot$. For instance, the reaction associating 1 to A, $\top$ to B, and $\bot$ to C is represented with $<$

[1] As explained in [19], monotony can be relaxed into *predictability*, which allows for non-deterministic internal choices of the component as long as they are published through the outputs, allowing the environment to change its behavior accordingly. But we will not cover this aspect here.

$A = 1, B = \top >$. 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 synchronous 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 non-void, finite, and disjoint sets of input and output signals, $\mathcal{S}$ is the set of states, and $TF: \mathcal{S} \times \mathcal{V}_\mathcal{I} \longrightarrow\!\!\!\!\circ\; \mathcal{S} \times \mathcal{V}_\mathcal{O}$ is the function representing the transitions. The function is partial to represent the fact that the system may not accept any input.*

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 our specifications throught this paper.

A Mealy machine can stutter in state s if the *stuttering transition* $s \xrightarrow{/} s$ is defined.

We say that a Mealy 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. To facilitate the definitions and presentation of this paper, we require all synchronous automata to have no input-less transition (but similar results exist for deterministic systems with input-less transitions).

2.3 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 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,

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

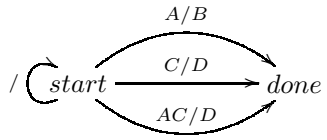


Figure 1: The PREEMPT example (top), and its Mealy machine representation (bottom)

depending on whether S is an output or an input signal. We shall 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 (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. 1. 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 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 such synchronous programs, 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 deterministic asynchronous implementations without adding such new messages to encode absence. We then advocate for a 2-step implementation process, where all absence encoding problems are dealt with *inside the synchronous model*, thus facilitating the analysis of large specifications:

Step 1: Signal absence encoding. Transform the synchronous specification into one where reaction to signal absence is not needed. 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.

Our characterization has important practical consequences, establishing the limits of the two-phase approach and allowing the choice of the best absence encoding.

2.4 Related work

Our work is closely related to work on various variants of *endochrony*, and we show 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*. Moreover, our two-step implementation process reflects the process of the Polychrony [10] environment developed around the Signal language.

Our work generalizes previous work on *weakly endochronous systems* by Potop, Caillaud, and Benveniste [19, 18]. The main difference is that we do not assume here all the goals of the approach based on weak endochrony and weak isochrony. More precisely, we do not address the preservation of the synchronous *composition* semantics throughout desynchronization. In particular, the decomposition into primitive reactions (extensively used in the proofs of [19]) will not be ensured in our approach, so that the characterization is closer to that of *microstep weak endochrony* [18].

Along the same lines, our work extends or is closely related to work on the various variants of *endochrony*. Compared with the endochronous systems of Benveniste, Caillaud, and Le Guernic [3], which is used in the compilation of the Signal language [1], our work generalizes by allowing concurrent reactions and a more relaxed input reading policy (more informations in Section 4.1). We already explained, in Section 2.1, that LeGuernic, Talpin, and Le Lann [10] take a *don't care*-based approach, different from ours.

The *latency-insensitive systems* of Carloni, McMillan, and Sangiovanni-Vincentelli, the Lustre language [12], and the AAA/SynDEx methodology [9] 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 [20] try to relax these constraints.

Our results do not cover the distributed implementation of reactive systems, as do Caspi, Girault, and Pilaud [7], nor have the same global approach. We only deal with the construction of one deterministic asynchronous component from one synchronous component.

From another perspective, the synchronous systems, as defined in distributed computing [2] correspond in our case to 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 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. Stuttering-invariance and determinism imply that the programs 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).[2] 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:

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

This program obviously has input-less transitions, because it can produce `O` without reading a single input (`TICK` is not an input signal). However, meaningful classes of Esterel programs do not have input-less transitions. Among them, all those containing no `pause` or `suspend` statements and no reference to `TICK`, and where all preemption triggers are reduced to one signal (without `not` operator).

To represent such behaviors with deterministic Signal programs (which have no input-less transition), the TICK signal must be explicitly represented in the input interface of the program, for instance:

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

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

Line 3 specifies that the state is read and updated in all reactions (*i.e.* 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) 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 starting in s $s \xrightarrow{r_k=i_k/o_k} s_k$, $r_k \neq /$, $k = 1, 2$, *we have:*

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

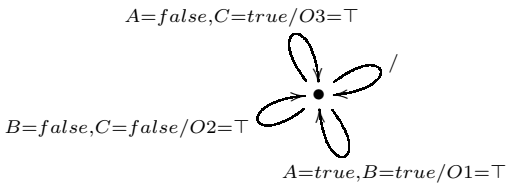
In other words, we can decide which transition to do by testing the value (and not the presence/absence) of an input. This choice can be implemented as a deterministic choice in our asynchronous framework.

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

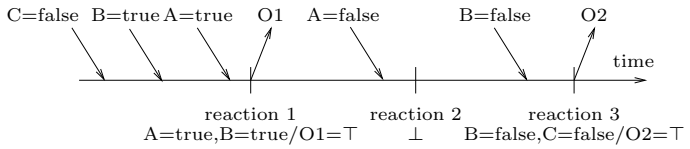


Figure 2: Incremental ASAP asynchronous execution of NOABSENCE1

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 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. 2. 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 `C`. 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 FIFOs of `A` and `B` unblocked (new messages can be accepted), but the FIFO of `C` remains blocked by the unconsumed message on it. 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 $B = false, C = false/O2 = \top$.

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 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. 2, for instance, the input reading process $Wait_{B=false,C=false}$ starts when execution starts. Then, it assembles $C = false$, but $B = true$ arrives and $Wait_{B=false,C=false}$ is be 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.

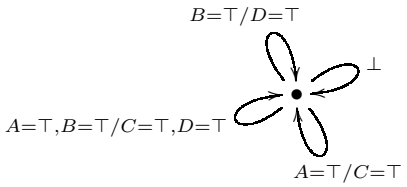
Incremental ASAP input reading is more complex than endochronous input reading but allows the deterministic asynchronous implementation of more synchronous systems. The basic Incremental ASAP technique described here can be optimized, *e.g.* by using blocking reads (like in the endochronous approach) 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 [19] 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 Concurrency and 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 need to add two new rules allowing the Incremental ASAP policy to handle 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 $supp(i_1) \cap supp(i_2) \neq \emptyset$. When a clock activation arrives, the following supplementary rules apply:

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 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.

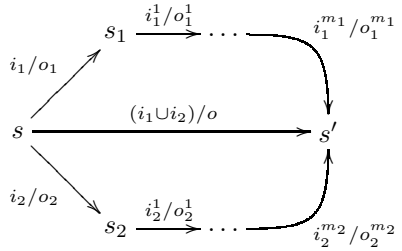
These properties are implied by the NRSA property, and need to be preserved by its extension to concurrent systems.

5.2 Concurrent NRSA

Consider a synchronous specification $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$ with the property that its implementation using a Concurrent Incremental ASAP input reading policy produces a monotonous and deterministic[3] asynchronous automaton.

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 $o_1 \cup o_2 \subseteq o$.

From the confluence property we demanded for the asynchronous implementations in Section 2.1, 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):

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

$$o = o_1 \cup \bigcup_{j=1}^{m_1} o_1^j = o_2 \cup \bigcup_{j=1}^{m_2} o_2^j \tag{2}$$

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),(1), and (2).*

[3] In the sense of Kahn [13].

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. Then, the Concurrent Incremental ASAP asynchronous implementation of Σ is monotonous and deterministic if and only if Σ has the Concurrent NRSA property.*

Proof sketch: $\Rightarrow$. Already done, in a constructive fashion, in this section.
$\Leftarrow$. Consider a finite set of inputs (a sequence of non-absent values for each input signal), consider two arrival orders of these incoming inputs with respect to the clock triggering instants, and consider the associated maximal Concurrent Incremental ASAP executions. Given that every transition (stuttering ones excepted) consumes at least one input, the two executions are finite.

The next step is to prove that the two executions read the same inputs. By assuming that one execution reads one more input than another on some input channel, and by using equation (*cnrsa*), we contradict the maximality hypothesis on the executions. This reasonment also proves that the destination state is the same.

The last step is to prove that the same outputs are produced on both execution paths. Similarly, by using the Concurrent NRSA property, the rewriting is confluent. **End proof.**

5.2.1 Relation with weak endochrony

Consider a synchronous system Σ satisfying the Concurrent NRSA property and two non-contradictory transitions $s \xrightarrow{i_k/o_k} s_k$, $k = 1, 2$. 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 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 $o \setminus o_1$.

If we restrict our class of specifications to those where the input arrival order does not lead to an explosion in the number of clock cycles needed for consuming them, we can further require that once a reaction i/o is fireable in a state, and as soon as the inputs of i arrive through the input FIFOs, all the inputs of i are consumed and all the outputs of o are produced. In this class of specifications, those that have deterministic asynchronous implementations are characterized by property (*cnrsa*), with the restriction that $m_1, m_2 \leq 1$ (the diagram (*cnrsa*) closes in at most one step). This property is similar to the microstep variant of weak endochrony defined by Potop and Caillaud [18].

Similarly, if we require that rule SR1 is never applied, which greatly simplifies the structure of the asynchronous implementation, we are lead to a decomposition of all executions into atomic reactions. The result is macrostep weak endochrony [19].

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 necessary in the deterministic asynchronous implementation of a synchronous system.

6.1 A simple example in Esterel, Signal, and Lustre

Consider the Esterel example `PREEMPT`, of Fig. 1. We remind the program body:

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

We explained in section Section 2.3 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 `B`) 2 different outputs can be obtained.

The 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

Other properties that introduce encoding problems in the asynchronous implementation of synchronous specifications can be traced back to reaction to signal absence.

First of all, 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 $TICK = \top/$, $TICK = \top, A = \top/$, $TICK = \top, B = \top/D = \top$, and $TICK = \top, A = \top, B = \top/D = \top$.

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

```
process LOSS1 =
(? event A,B,TICK ; ! event C,D ;)
(| A ^< TICK | B ^< TICK
 | AE ^= BE ^= stateA ^= stateB ^= TICK
 | AE := (true when A) default false
 | BE := (true when B) default false
 | stateA := stateAnxt $ init true
 | 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 an 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.

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] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):18, Sep 2001.

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

[8] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, pages 226–238, 1996.

[9] 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.

[10] 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.

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

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

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

[14] R. Keller. A fundamental theorem of asynchronous parallel computation. *Lecture Notes in Computer Science*, 24:103–112, 1975.

[15] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical report, DAIMI, Arhus University, 1977.

[16] M. Nebut. *Réactions synchrones: spécification et analyse (French for: Synchronous reactions: specification and analysis)*. PhD thesis, University of Rennes 1, France, nov 2002.

[17] M. Nebut. Specification and analysis of synchronous reactions. *Formal Aspects of Computing*, 16(3):263–291, august 2004.

[18] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proccedings ACSD'05*, St. Malo, France, June 2005.

[19] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006. First published in Proceedings ACSD'04.

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

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399