INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Deterministic execution of synchronous programs in an asynchronous environment

A compositional necessary and sufficient condition

Dumitru Potop-Butucaru — Robert De Simone — Yves Sorel

N° 6656

Avril 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6656--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Deterministic execution of synchronous programs in an asynchronous environment

A compositional necessary and sufficient condition

Dumitru Potop-Butucaru , Robert De Simone* , Yves Sorel

Thème COM — Systèmes communicants
Équipe-Projet AOSTE

Rapport de recherche n° 6656 — Avril 2008 — 20 pages

Abstract: Synchronous reactive formalisms form an appealing programming model for embedded system and Systems-on-Chip (SoC) design. Deploying synchronous programs onto asynchronous distributed execution platforms is an important issue, and has been the topic of substantial research in the past. The point is that signal/event absence in a reaction cannot be taken as granted because of communication latencies. A simple solution consists in systematically sending signal absence notifications, but it is unduly expensive at run-time. Sufficient properties have been proposed defining subsets of synchronous programs where asynchronous evaluation is faithful to their original specification. In essence they aim at preserving stream computation *monotonicity*, in the original formulation of Kahn Network principles, or *confluence*, as coined by R. Milner. Some of these criteria may become quite involved. In the current paper we show a precise technical result: If equivalence between the synchronous and the asynchronous semantics is congruence with respect to parallel constructors, then the ”good” criterion amounts to a single step ”diamond closure” property, with independent behaviors converging to the union of their effects. It should be remembered here that the *local* individual behaviors of components may themselves contain simultaneous events, thereby allowing complex synchronous modeling on this lower layer.

Key-words: synchronous specification, asynchronous implementation, execution machine, determinism, Kahn principle, monotony, confluence, desynchronization, necessary and sufficient condition

* UR de Sophia Antipolis-Méditerranée

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Execution déterministe de programmes synchrones dans un environnement asynchrone

Une condition compositionnelle nécessaire et suffisante

Résumé : Les formalismes réactives synchrones fournissent un modèle adapté à la programmation de systèmes embarqués et de systèmes sur puces. L'implantation de programmes synchrones sur plates-formes d'exécution asynchrones, potentiellement distribuées, est donc un problème important, qui a déjà fait l'objet d'importants efforts de recherche. Sur de telles plates-formes, les latences non-bornées des communications font que l'absence d'un signal/événement dans une réaction ne peut pas être testée. Pour simplement résoudre ce problème, on peut toujours envoyer les notifications d'absence, mais cela peut s'avérer très coûteux. Une meilleure solution consiste dans la définition de conditions assurant que l'évaluation asynchrone d'un programme synchrone préserve la sémantique synchrone initiale. En principe, ces conditions suffisantes visent à assurer la *monotonie* de la fonction d'entrée/sortie de l'implantation asynchrone (dans la formulation de G. Kahn) , ou sa *confluence* (dans la formulation de R. Miller). Dans cet article, nous proposons un résultat technique: Si l'équivalence entre la sémantique synchrone et celle asynchrone est la congruence par rapport aux opérateurs de composition parallèle, alors un critère nécessaire et suffissant est donné par une simple "propriété de losange" où les comportements indépendants convergent vers l'union de leurs effets.

Mots-clés : spécification synchrone, implantation asynchrone, machine d'exécution, déterminisme, principe de Kahn, monotonie, confluence, désynchronisation, condition nécessaire et suffisante

1 Introduction

Synchronous reactive formalisms [8, 3] 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 propagation 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* [2, 7, 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 (as coined by R. Milner [12]), so that the first one does not discard the second. This is also linked to the Kahn principles for networks [10], where only internal choice is allowed to ensure that overall lack of confluence cannot be caused by input signal speed variations.

Contribution We formally define a general execution machine for synchronous programs over an asynchronous environment. Execution is based on (1) transforming the explicit "absent" values of the synchronous model into actual absence of messages in the asynchronous environment and (2) triggering a synchronous reaction as soon as enough inputs have arrived through the asynchronous communication lines.

We then characterize the synchronous programs give monotonous and deterministic asynchronous systems (in the sense of the Kahn process networks) when run over our execution machine.

Most important, we also determine that there exists a unique greatest sub-class of such programs that is closed under synchronous composition. The characterization is given by a simple diamond closure property.

This simple characterization and the unicity result are important, because they offer a good basis for the development of similar criteria at the level of various synchronous languages and formalisms (or the definition of language sub-sets where some restricted form of concurrency ensures this property by construction).

Our characterization is also important because it corresponds to a very general execution mechanism covering current practice in embedded system design. Thus, it fixes theoretical limits to current implementation techniques.

Outline The remainder of the paper is organized as follows: Section 2 defines our model of synchronous specification, the desired Kahn-like asynchronous implementations, gives a short motivation, and reviews related work. Section 3 defines our im-

plementation technique, which includes a definition of the needed execution machines, with full detail. In Section 4 we specialize onto the chosen class of implementations the desired Kahn-like correctness properties. Section 5 derives the main result of the paper. We conclude in Section 6. **Due to space limitations, all proofs are grouped in the appendix.**

2 Basic Notions

In this section, we introduce the concepts and formal definitions that will be used throughout the paper, and we intuitively define our problem.

2.1 General notations

Given a set $\mathcal{S}$, we denote with $\mathcal{S}^*$ the set of all finite words over $\mathcal{S}$, and with $\mathcal{S}^\omega$ the set of all finite and infinite words over $\mathcal{S}$. We denote with ϵ the empty sequence, and with $h[i]$ the i^{th} value of a sequence h, for $i \geq 1$ (instead of a single index we also allow the use of a *range* $1..n$ to identify a sub-sequence). Sequences are partially ordered by the prefix order $\preceq$. We say that two sequences $h_1, h_2 \in \mathcal{S}^\omega$ are non-contradictory, denoted $h_1 \bowtie h_2$ when one is a prefix of the other (in any order). We denote with $\vee$, respectively $\wedge$ the least upper bound and greatest lower bound operators induced by $\preceq$. The concatenation of two sequences h_1 and h_2 is denoted $h_1.h_2$ (h_1 must be finite). When $h_1 \preceq h_2$ we denote with $h_2 \setminus h_2$ the sequence such that $h_1.(h_2 \setminus h_1) = h_2$. Any increasing sequence $(h_j)_{j=1}^{\infty}$ of $\mathcal{S}^\omega$ has a limit $\lim_{j\to\infty} h_j$.

The prefix order on individual ${\mathcal{S}_i}^\omega$ induces component-wise a prefix order on the product set $\prod_{i=1}^{n} {\mathcal{S}_i}^\omega$. Similarly, the $\bowtie$ relation, and the $\vee$, $\wedge$, concatenation and limit operators are extended component-wise on the product set.

2.2 Synchronous Mealy Machine

The various synchronous formalisms [8] are used to develop specifications that can be interpreted as *finite-state incomplete synchronous Mealy machines*. This is the model we use throughout the paper to represent synchronous *programs* or *hardware components*.

In a synchronous Mealy machine, the transitions are labeled with *reactions*. An *execution (trace)* is a sequence of reactions indexed by the *global clock*.

A reaction is a valuation of the *input and output signals* of the synchronous machine. 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 $\mathcal{D}_S^\perp$. When we are only interested in the presence or absence of a signal S, we use a special domain $\mathcal{D}_\top$ that has only one value $\top$ (present).

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. Given

$r \in \mathcal{R}(X)$ and a set of signals X', we denote with $r\ |_{X'}$ the image of r through X' (which equals r on $X \cap X'$ and $\bot$ on $X' \setminus X$).

To represent reactions in a compact form, 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 $< A = 1, B = \top >$. The delimiters can be dropped if there is no confusion. On non-contradictory reactions we define the union ($\cup$), intersection ($\cap$), 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.

Definition 1 (incomplete Mealy machine) *An incomplete synchronous Mealy machine is a tuple $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, s_0, \mathcal{T})$, where $\mathcal{I}$ and $\mathcal{O}$ are the finite and disjoint sets of input and output signals, $\mathcal{S}$ is the set of states, $s_0 \in \mathcal{S}$ is the initial state, and $\mathcal{T} : \mathcal{S} \times \mathcal{R}(\mathcal{I}) \longrightarrow\!\!\!\!\!\circ\ \ \mathcal{S} \times \mathcal{R}(\mathcal{O})$ is the partial transition function.*

Using partial transition functions is common in synchronous system design, and amounts to defining the admissible behaviors of both the system and its environment. Note that the functional definition of the transitions implies *determinism*,[1] a property we require for all synchronous modules throughout this paper.

We will write $s \xrightarrow[\Sigma]{i/o} s'$ or $s \xrightarrow{i/o} s'$ instead of $\mathcal{T}(s, i) = (s', o)$ to represent system transitions. We denote with $Traces_\Sigma(s) \subseteq \mathcal{R}(\mathcal{I} \cup \mathcal{O})^\omega$ the set of traces of the synchronous module Σ starting in $s \in \mathcal{S}$. The determinism of Σ implies that for all finite trace $t \in Traces_\Sigma(s)$ we can define the (unique) destination state, denoted $s.t$. In this case we also write $s \overset{t}{\Longrightarrow} s.t$. We denote $RSS(\Sigma) = \{s_0.t \mid t \in Traces_\Sigma(s_0)\}$.

A transition $s \xrightarrow{i/o} s'$ is called an *input-less transition* if $i = \bot$. A Mealy machine can *stutter* in state s if the loopback *stuttering transition* $s \xrightarrow{/} s$ assigning $\bot$ to all signals is defined. In this case, we will say that s is a *stuttering state*. We say that a machine is *stuttering-invariant* when it can stutter in any state. Such a machine can spend time (reactions) in any state doing nothing (*e.g.* waiting to synchronize with other machines). Given a trace $t \in Traces_\Sigma(s)$, we shall denote with $\overline{t} \in Traces_\Sigma(s)$ its *normal form* with no stuttering transitions.

To simplify subsequent definitions and proofs and focus on the synchronous/asynchronous interface, we shall restrict our investigation to Mealy machines Σ that have no infinite sequence of input-less transitions different from the stuttering ones. However, our results can be extended to deal with infinite sequences of input-less transitions.

2.3 Kahn process networks

In 1974, Gilles Kahn wrote his seminal paper [10] 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

[1] At most one transition for given state and input. This requirement means that we cannot model sensors.

until a message is available. There is no block on writing. Once sent, a message reaches its destination in a finite (but unbounded) time. 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 mechanism exists between processes (which run in parallel, asynchronously). 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* modeling of deterministic systems that operate infinitely using limited resources [11]. Variants or extensions of the original model are currently used in a variety of academic and industrial settings.

2.3.1 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. Given an execution of the KPN, the *history* $Hist(c)$ of a channel c is the sequence of all messages that are placed on c during the execution. For a finite execution, $Hist(c) \in {\mathcal{D}_c}^*$. For an infinite execution, $Hist(c) \in {\mathcal{D}_c}^\omega$.

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

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

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

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

2.3.2 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 cannot result in less output messages.
- *Continuous*, in the sense of the limit operator, meaning that for any sequence of input histories $h_k \in \prod_{j=1}^{n^f} {\mathcal{D}_{ci_j^f}}^\omega, k \geq 1$ such that $\lim_{k\to\infty} h_k = h$ we also have $\lim_{k\to\infty} f(h_k) = f(h)$.

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:

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

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

2.4 Motivation

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. **We focus on the problem of constructing one asynchronous process with monotonous and continuous stream function from one concurrent synchronous module**. The Kahn principle can then 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 distributed implementation of a synchronous specification. We only focus here on the interface between the synchronous and the asynchronous domains of a single process.

2.5 Related work

Our results are closest related to results on the implementation of synchronous languages and formalisms for execution in an asynchronous environment. The standard simulators generated for Esterel[4] and Lustre[9] take the approach of reading and sending the status of all inputs and outputs at all reactions. The same approach is taken in the theory of *latency-insensitive systems*[5] to ensure that the initial synchronous semantics is preserved after the synchronous communication lines are replaced with FIFOs of unspecified latency.

The development of the Signal/Polychrony language has lead to a series of results on the (distributed) asynchronous implementation of a synchronous specifications. Several criteria have been proposed to identify programs that give monotonous and continuous implementations. We cite here (1) the first proposed criterion *endochrony*[2], which did not allow concurrent systems and was non-compositional, (2) the compositional *weak endochrony*[13], which generalizes to a synchronous setting the classical theory of Mazurkiewicz traces, and (3) the easily-checkable criteria derived to allow the analysis of Signal/Polychrony programs [15, 1].

Our work also seems closely related to results concerning the design of asynchronous [6] and burst-mode [16] circuits. Our monotony concerns, in particular, can be seen as a speed independence property guaranteeing correctness and determinism regardless of the relative speeds of different computations and communications.

3 GALS Implementation Technique

In this section we define our implementation technique: Implementation structure, signal absence encoding, and ASAP execution policy.

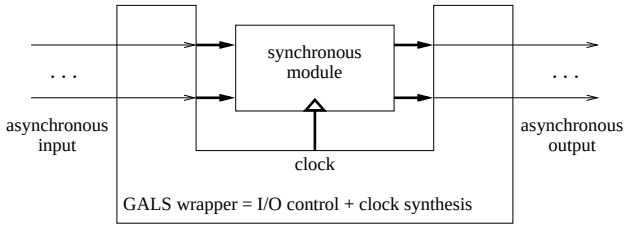


Figure 1: Desired GALS implementation structure. The synchronous module can be software (a reaction function) or hardware (a circuit).

3.1 Target implementation structure

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*). The synchronous module can be a sequential reaction function (in software), or a synchronous digital circuit (in hardware). The execution machine drives the synchronous module in the asynchronous framework defined above by performing:

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

This general pattern covers a large class of implementations.

3.2 Signal absence encoding

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 (i.e. no message at all). Unfortunately, for-

getting 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.

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

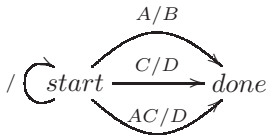


Figure 2: A small Esterel[4] program (left), and its Mealy machine representation (right)

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, we determine which synchronous programs give deterministic implementations while encoding absent values with message absence.

3.3 ASAP reaction triggering

A second assumption we make is that the asynchronous wrapper triggers a synchronous reaction as soon as its inputs are available on the input channels.

```
process NOABSENCE1=(?boolean A, B, C;
                    ! event X, Y, Z;)
(| X ^= when A=true ^= when B=true
 | Y ^= when B=false ^= when C=false
 | Z ^= when A=false ^= when C=true |)
```

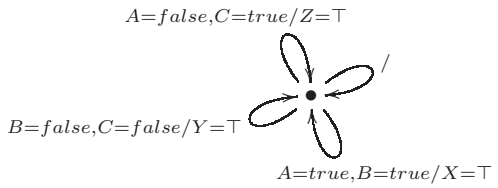


Figure 3: A small Signal/Polychrony[7] program (left), and its Mealy machine representation (right)

The ASAP reaction triggering policy means that a reaction i/o can be triggered whenever the present values of i are all available as messages on the corresponding input FIFOs. 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.

We exemplify on the program of Fig. 3. One possible asynchronous execution is given in Fig. 4. It corresponds to the case where reactions are triggered by an external clock. The input FIFO associated with signal C is the first to deliver a value (false).

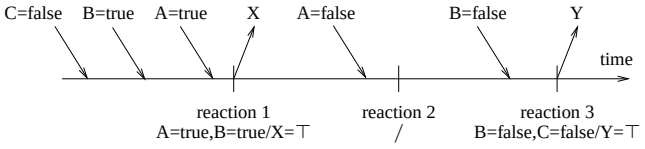


Figure 4: Incremental ASAP asynchronous execution of NOABSENCE1

Then, new values arrive for B and A. When a reaction is triggered by the external clock, the only non-stuttering fireable reaction is $A = true, B = true/X = \top$. This reaction is performed, X 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 $\perp$, which changes nothing. The third reaction is $B = false, C = false/Y = \top$.

A reaction is performed as soon as its inputs are available at clock activation time. 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)*.

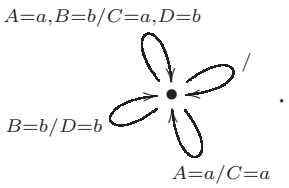
We assume that no computation cycle is triggered when no reaction is fireable given the current input. This assumption can be satisfied either through constraints on the synchronous module (*e.g.* stuttering-invariant modules always have the fireable stuttering reaction), or by employing specific clock triggering policies, such as *pausible clocking*[17].

3.3.1 Non-determinism. Fairness.

The example above has the nice property that at most one non-stuttering reaction is fireable at all instants, because the non-stuttering reactions are mutually contradictory. Thus, no ambiguity exists as to which transition should be triggered at clock activation time.

However, simple Mealy machines do not satisfy this hypothesis. Consider the example of Fig. 5. When values arrive for both A and B between two clock activations,

```
process PAR = (? boolean A, B ;
               ! boolean C, D ;)
(| C := A | D := B |)
```



.

Figure 5: A simple parallel composition (left), and its Mealy machine representation (right)

the 3 non-stuttering transitions become fireable. To cover such cases, we allow the

non-deterministic triggering of any one of the fireable reactions (including the stuttering one).

Non-determinism implies the need for *fairness*, to prevent starvation. We require our ASAP execution machine to satisfy the following fairness constraint: A state s cannot be traversed infinitely many times during an execution with transition $s \xrightarrow{i/o} s'$ fireable, and without the transition being taken at least once. This models the assumption that choice between fireable transitions in a state is random (so that the probability of taking a transition that is fireable infinitely many times is 1).

The non-deterministic modeling of the execution machines generalizes deterministic execution rules, such as the one of [14] and imposes no constraint on the executed synchronous module. The results we derive in this general model do apply to the deterministic cases, too.

3.3.2 Causality.

Note that the execution machine we defined enforces the rule that all inputs must arrive before all outputs in each reaction. This may restrict the synchronous causality of the original synchronous program, but the approach corresponds to existing practice in synchronous program implementation.

4 Implementation behavior and correctness

In this section we formally model the behavior of GALS implementations and specialize at their level the Kahn-like correctness properties we expect. To take into account the fairness hypothesis in a simple way, we cover here the case of finite executions. After determining the needed criteria in the finite case, we will show in Section 5.4 that they also imply monotony and continuity in the case of infinite executions.

4.1 Desynchronization operator

Given the synchronous machine $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T})$, we will denote with $[\Sigma]$ its GALS implementation. In the absence of a time reference, we use histories to model the sequences of messages arriving to or emitted by $[\Sigma]$ from/to the asynchronous environment.

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

4.2 Behavior

Consider the implementation $[\Sigma]$ of Σ, a state $s \in \mathcal{S}$, and $\chi \in \prod_{S \in \mathcal{I}} {\mathcal{D}_S}^{*}$ a finite input history. The fairness hypothesis considered on the execution machine ensures that $[\Sigma]$ will not stutter indefinitely in a state when some other transition is fireable

under available input. Also recall the hypothesis that no infinite sequence of input-less transitions exists in Σ (modulo stuttering). We deduce that any fair execution of $[\Sigma]$ with finite input χ will consume and produce all its inputs and outputs in a finite number of transitions, leading the synchronous module to a state where it can only stutter (no transition allowing consumption of remaining inputs, if any, or production of new outputs).

Recall that executions of $[\Sigma]$ are traces of Σ. Then, we can represent the behavior of $[\Sigma]$ as:

$$[\Sigma] : \mathcal{S} \times \prod_{S \in \mathcal{I}} {\mathcal{D}_S}^{\omega} \rightarrow \mathcal{P}\left(Traces_{\Sigma}(s)\right)$$

associating to each state s and finite input χ the set $[\Sigma](\chi, s)$ of all finite traces $t \in Traces_{\Sigma}(s)$ with $\delta(t) \mid_{\mathcal{I}} \preceq \chi$ which are maximal in their stuttering-free normal form $\overline{t}$ when compared to the other traces feasible under χ. This definition means that we only represent maximal behaviors, and we also overlook infinite fair traces obtained by appending an infinite sequence of stuttering transitions to one of $[\Sigma](\chi, s)$.

4.3 Correctness

Our objective is to obtain implementations $[\Sigma]$ that are representable as monotonous and continuous stream functions in the sense of Kahn. To allow representation as a stream function, all the behaviors of the implementation (starting in the initial state) for given input history must read and produce the same inputs and outputs. Formally:

$$\textbf{StreamFunc} : \forall s \in RSS(\Sigma) : \forall \chi_{\mathcal{I}} \in \prod_{S \in \mathcal{I}} {\mathcal{D}_S}^* : \exists \chi \in \prod_{S \in \mathcal{I} \cup \mathcal{O}} {\mathcal{D}_S}^* : \forall t \in [\Sigma](\chi_{\mathcal{I}}, s) : \delta(t) = \chi$$

When this property is satisfied, we can define the *stream function* of $[\Sigma]$ in state $s \in RSS(\Sigma)$ by $SF_{\Sigma,s}(\chi_{\mathcal{I}}) = \chi \mid_{\mathcal{O}}$. Property StreamFunc also implies that $SF_{\Sigma,s}()$ is monotonous, as more input invariably results in more output.

In addition to this, we will require our implementations to be *state deterministic*, meaning that all traces in $[\Sigma](\chi_{\mathcal{I}}, s_0)$ have the same destination state. Formally:

$$\textbf{StateDeterm} : \forall s \in RSS(\Sigma) : \forall \chi_{\mathcal{I}} \in \prod_{S \in \mathcal{I}} {\mathcal{D}_S}^* : \forall t_1, t_2 \in [\Sigma](\chi, s) : s.t_1 = s.t_2.$$

We denote with *NonComp* the class of synchronous machines satisfying Properties StreamFunc and StateDeterm.

5 Criteria for correct implementation

Properties StreamFunc and StateDeterm characterize *NonComp* in a way that does not offer good support to automated analysis (due to quantification over input histories). In this section, we derive simpler, equivalent criteria.

5.1 Non-compositional criterion

To determine a first, non-compositional criterion, we generalize the reasoning of [14] to cover the more general execution machine defined here.

Theorem 1 (Correctness criterion, non-compositional) *The synchronous Mealy machine Σ satisfies Properties StreamFunc and StateDeterm if and only if for all $s \in \mathcal{S}$ and $s \xrightarrow{i_k/o_k} s_k$, $k = 1, 2$, if $i_1 \bowtie i_2$ then there exists $t_k \in Traces_\Sigma(s_k)$ finite, $k = 1, 2$ with $s_1.t_1 = s_2.t_2$, $\delta((i_1/o_1).t_1) = \delta((i_2/o_2).t_2)$, and $\delta((i_1/o_1).t_1) \mid_{\mathcal{I}} = \delta(i_1 \cup i_2)$. In other words, we have confluence.*

This theorem gives a simpler characterization to *NonComp*, one that can be checked in finite time for a finite Mealy machine. However, we are still far from a practical characterization because the confluence occurs in an unspecified number of steps. Moreover, we will see in the following sections that the property is not preserved by composition, which makes it less appealing for use with a practical (incremental) systems construction methodology.

5.2 Synchronous composition

In defining synchronous composition, we introduce two limitations that allow us to remain focused on our synchronization-oriented asynchronous implementation problem: (1) We only consider acyclic networks of interconnections, and (2) We require that the output sets of the composed systems must be disjoint, so that we don't need to enforce the coherency of outputs (e.g., same value on the same signal).

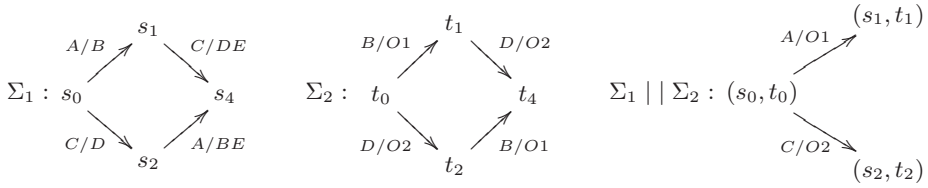
The first requirement allows us to avoid complex causality issues without giving up causality altogether, like in [2, 13]), or defining complex causality-related aspects that do not fit this paper. We can remain at the chosen description level, and the results can be easily generalized for more complex composition mechanisms.

Definition 2 (composability and composition) *Consider two synchronous modules $\Sigma_i = (\mathcal{I}_i, \mathcal{O}_i, \mathcal{S}_i, \mathcal{T}_i, s_i^0)$, $i = 1, 2$. We say that Σ_1 and Σ_2 are* composable, *in this order, if $\mathcal{I}_1 \cap \mathcal{O}_2 = \emptyset$ and $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$ (recall that $\mathcal{I}_i \cap \mathcal{O}_i = \emptyset$, from the definition of synchronous automata).*

If Σ_1 and Σ_2 are composable, then their composition is the synchronous module $\Sigma_1 \,||\, \Sigma_2 = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{T}, s^0)$ with $\mathcal{I} = \mathcal{I}_1 \cup (\mathcal{I}_2 \setminus \mathcal{O}_1)$, $\mathcal{O} = \mathcal{O}_2 \cup \mathcal{O}_1$, $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$, $s^0 = (s_1^0, s_2^0)$, and where the transition $(s_1, s_2) \xrightarrow[\Sigma_1 | \, |\Sigma_2]{i/o} (s_1', s_2')$ exists if an only if there exist (unique, due to determinism): $s_1 \xrightarrow[\Sigma_1]{i|_{\mathcal{I}_1}/o_1} s_1'$, $s_2 \xrightarrow[\Sigma_2]{o_1|_{\mathcal{I}_2} \cup i|_{\mathcal{I}_2}/o|_{\mathcal{O}_2}} s_2'$ with $o = o_1 \mid_{\mathcal{O}} \cup o_2 \mid_{\mathcal{O}}$.

5.3 Compositional criterion

Note that the correct implementation criteria of Theorem 1 are not preserved by composition. A simple example:



Now, we seek to determine large sub-classes of *NonComp* that are closed under synchronous composition. In doing so, it is important to see that doing this is interesting only when the new classes of modules includes meaningful ones.

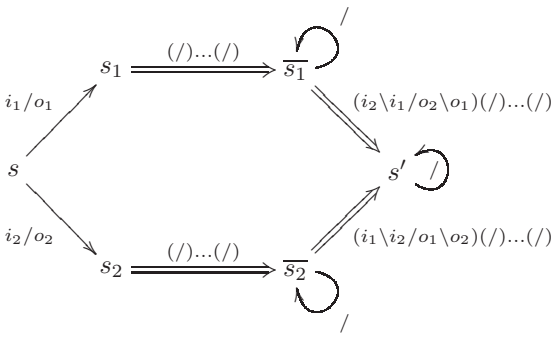
In particular, we consider such a sub-class must include simple sequential behaviors: finite sequences of transitions with stuttering transitions in each state. These synchronous machines have no choice, nor concurrency, nor infinite stuttering-free executions. We denote with *Simple* this sub-class synchronous systems. All modules of *Simple* satisfy our correct implementation criteria, and so do all compositions of such modules.

In this section we prove that there exists a unique largest sub-class of *NonComp* including *Simple* and closed under synchronous composition. Moreover, this class, denoted *Comp*, has a simpler characterization.

Theorem 2 (characterization) *Let $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, s_0, \rightarrow)$ be a synchronous module. Assume of $\Sigma \in$ NonComp, and that any finite composition involving Σ and programs of the Simple class is also in NonComp. Then, for all $s \in RSS(\Sigma)$:*

Stuttering: *There exists a unique $\overline{s}$ such that $s \xRightarrow{(/)\ldots(/)} \overline{s}$ with a / self-loop (and there are no other input-less transitions, due to determinism). This existence result defines a new operator $\overline{s}$ we use in the remainder of the paper.*

Diamond: *If $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$ with $i_1 \bowtie i_2$, then $o_1 \bowtie o_2$ and there exists a unique s' such that:*



where $(/)\ldots(/)$ denotes finite sequences of void transitions.

We denote with *Comp* the class of modules satisfying Properties Stuttering and Diamond. We prove *Comp* is the class we are looking for.

To do this, we only need to prove closure under synchronous composition.

Theorem 3 (compositionality) *The class Comp is closed under synchronous composition.*

These two theorems complete our construction, and we have the following corollary:

Corollary 1 (compositional criterion) *The class Comp is the largest sub-class of NonComp including Simple and closed under synchronous composition.*

It is interesting to note that for any $\Sigma \in$ *Comp*, there exists one that gives the same stream functions, yet has no void transition that is not stuttering. Such a synchronous module is easily obtained by unifying each state s in Σ with $\overline{s}$. The result of this quotient operation, denoted $\overline{\overline{\Sigma}}$, can be seen as a normal form of Σ and satisfies a simpler "diamond closure" property, which is also compositional. However, the non-stuttering void transitions of Σ may be useful to express best-case timing properties that obey max+ rules upon composition.

5.4 The infinite case

The only thing that remains to be done is to prove that the fair execution of synchronous modules of *Comp* gives monotonous and continuous stream functions (which includes the analysis of the case of infinite input histories). To do so, we prove the following theorem.

Theorem 4 (Infinite case) *Consider* $\Sigma \in$ *Comp*, $s \in RSS(\Sigma)$, *and* $\chi_{\mathcal{I}} \in \prod_{S \in \mathcal{I}} {\mathcal{D}_S}^{\omega}$. *Consider* $t \in Traces_{\Sigma}(s)$ *a maximal* fair *execution of* $[\Sigma]$ *under input* $\chi_{\mathcal{I}}$. *Then:*

$$\delta(t) = \lim_{\substack{\chi'_{\mathcal{I}} \to \chi_{\mathcal{I}} \\ \chi'_{\mathcal{I}} \textit{ finite}}} SF_{\Sigma,s}(\chi'_{\mathcal{I}})$$

This means that we can extend $SF_{\Sigma,s}()$ to a function on all histories, finite and infinite, that is monotonous and continuous.

6 Conclusion

We have defined a general execution machine for synchronous programs over an asynchronous environment. We determined which synchronous programs give monotonous and deterministic asynchronous systems (in the sense of the Kahn process networks) when run by our machine. We also determined that there exists a unique greatest subclass of such programs that is closed under synchronous composition.

The simple "diamond closure" characterization and the unicity of this class are important. First of all, they set the theoretical limits of current implementation techniques. Second, they offer a good basis for the development of similar criteria at the level of various synchronous languages and formalisms.

A first question is how to generate the synchronous machines of *Comp* using existing synchronous languages. Currently, this is not an easy task.[2] The dual problem is that of determining the sub-class of *Comp* can be programmed in existing languages.

From a practical point of view, we are interested in efficiently determining when a synchronous program belongs to *Comp* or some large sub-class. Maybe more interesting from the developer point of view, we are interested in defining language sub-sets where some restricted form of concurrency ensures this property by construction.

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] Esterel can't define complex environment constraints, which are essential in defining complex incomplete automata. Signal/Polychrony does not have an implicit global clock (it must be explicitly declared and related to all other signals).

[2] 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.

[3] 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.

[4] G. Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, Cambridge, MA, 2000. MIT Press.

[5] 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.

[6] J. Cortadella, M. Kishinevsky, A. Kondratiev, L. Lavagno, and A. Yakovlev. *Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[7] 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.

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

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

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

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

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

[14] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In *Proceedings EMSOFT'07*, Salzburg, Austria, October 2007.

[15] J.-P. Talpin, J. Ouy, L. Besnard, and P. L. Guernic. Compositional design of isochronous systems. In *Proccedings DATE'08*, Munich, Germany, 2008.

[16] K. Yun and D. Dill. Automatic synthesis of extended burst-mode circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):101–132, 1999.

[17] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)*, 1996.

APPENDIX: Proofs and lemmas

To prove Theorem 1, we shall actually prove the following result (of which variant (A) is the hypothesis and variant (C) is the conclusion of Theorem 1.

Theorem 5 (Correctness criterion, non-compositional) *Consider the synchronous module Σ. Then, the following statements are equivalent:*

A. *Σ satisfies Properties StreamFunc and StateDeterm.*

B. *For all $s \in RSS(\Sigma)$ and $t_1, t_2 \in Traces_\Sigma(s)$, finite, if $\delta(t_1 \mid_\mathcal{I}) \bowtie \delta(t_1 \mid_\mathcal{I})$, then there exists $t'_i \in Traces_\Sigma(s.t_i)$ finite, i=1,2 with:*

1. $s.t_1.t'_1 = s.t_2.t'_2$
2. $\delta(t_1.t'_1) = \delta(t_2.t'_2)$
3. $\delta(t_1.t'_1 \mid_\mathcal{I}) = \delta(t_1 \mid_\mathcal{I}) \vee \delta(t_2 \mid_\mathcal{I})$

C. *The same as (B), except that t_1 and t_2 are quantified over single transitions (traces of length 1).*

Proof sketch: $\mathbf{A} \Rightarrow \mathbf{C}$: Consider $t_j = i_j/o_j \in Traces_\Sigma(s), j = 1, 2$ in the hypothesis of (3). Let $\chi_\mathcal{I} = \delta(t_1 \mid_\mathcal{I}) \vee \delta(t_2 \mid_\mathcal{I})$. For $j = 1, 2$ let $t_j.t'_j$ be a stuttering-free maximal execution of $[\Sigma]$ under input history $\chi_\mathcal{I}$. Then, by applying the stream function hypothesis (Property StreamFunc), we obtain $\delta(t_1.t'_1) = \delta(t_2.t'_2)$. From the state determinism hypothesis (Property StateDeterm) we get $s.t_1.t'_1 = s.t_2.t'_2$. Also, given that $\delta(t_j \mid_\mathcal{I}) \preceq \delta(t_1.t'_1 \mid_\mathcal{I}) \preceq \chi_\mathcal{I} = \delta(t_1 \mid_\mathcal{I}) \vee \delta(t_2 \mid_\mathcal{I})$, $i = 1, 2$, we obtain the last property.
$\mathbf{C} \Rightarrow \mathbf{B}$: Statement (B) is easily proved by induction over the number of signal valuations in both t_1 and t_2.
$\mathbf{B} \Rightarrow \mathbf{A}$: The difficulty is the construction, for given s and $\chi_\mathcal{I}$, of χ and s'. To do this, consider the set $T(s, \chi_\mathcal{I})$ of all $t \in Traces_\Sigma(s)$, stuttering-free, such that $\delta(t \mid_\mathcal{I}) \preceq \chi_\mathcal{I}$. The set is finite. Say its elements are $t_1, \ldots, t_n$. Then, we can apply the hypothesis successively to:

- t_1 and t_2, to obtain t'_1 and t'_2
- $t_2.t'_2$ and t_3 to obtain t''_2 and t'_3
- $t_3.t'_3$ and t_4 to obtain t''_3 and t'_4

  $\ldots$

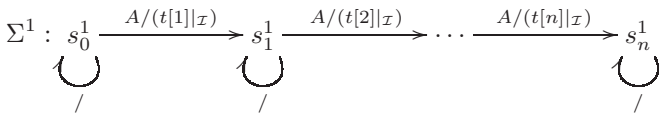
- $t_{n-1}.t'_{n-1}$ and t_n to obtain t''_{n-1} and t'_n

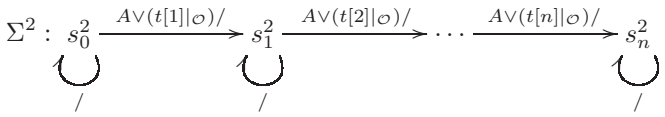
Then, $\chi = \delta(t_n.t'_n)$ and $s' := s.t_n.t'_n$ satisfy our needs. $\diamond$

Lemma 1 (composition) *The composition of two synchronous Mealy machines is a synchronous Mealy machine.*

Proof sketch: The non-trivial part of the proof is the determinism of $\Sigma_1 \mid\mid \Sigma_2$, which is due to acyclicity and determinism of the composed machines. $\diamond$

Proof sketch:**(of Theorem 2)** Consider $s \in RSS(\Sigma)$. Let $s_0 \stackrel{t}{\Longrightarrow} s$ be a trace leading to s of minimal length n. We denote with $s_k = s.t[1..k]$ for $1 \leq k \leq n$.
Property StreamFunc: We define the synchronous machines $\Sigma^i = (\mathcal{I}^i, \mathcal{O}^i, \mathcal{S}^i, s_0^i, \rightarrow^i)$, $i = 1, 2$ with: $\mathcal{I}^1 = \{A\}$, $\mathcal{O}^1 = \mathcal{I}$, $\mathcal{I}^2 = \mathcal{O} \cup \{A\}$, and $\mathcal{O}^2 = \emptyset$ with: $A \notin \mathcal{I} \cup \mathcal{O}$, $\mathcal{D}_A = \mathcal{D}_\top$ and transition relations defined by:

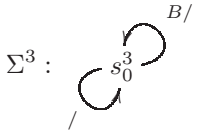




Then, the composition $\Sigma' = \Sigma^1 \,||\, \Sigma \,||\, \Sigma^2$ is defined.

Assume by absurd that Σ has no transition labeled with $/$ in state s. Then, there is no transition in the reachable state (s_n^1, s_n, s_n^2) of Σ'.

Consider now $\Sigma^3 = (\mathcal{I}^3, \mathcal{O}^3, \mathcal{S}^3, s_0^3, \rightarrow^3)$ defined by $\mathcal{I}^3 = \{B\}$, $\mathcal{O}^3 = \emptyset$, non-interferent with Σ', and with the transition relation:



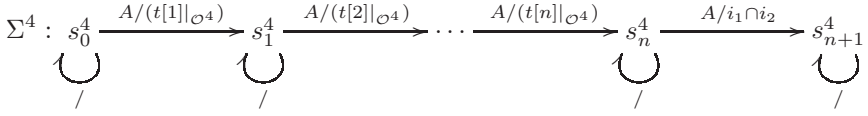
The composition $\Sigma' \,||\, \Sigma^3$ is defined and non-interferent, but according to our assumption by absurd it does not satisfy Properties StreamFunc and StateDeterm. This is due to the fact that when $\Sigma' \,||\, \Sigma^3$ reaches state (s_n^1, s_n, s_n^2) on its component Σ', execution is blocked. However, this may occur after various executions of the Σ^3 component, meaning that various amounts of B signals are consumed, and confluence is not possible.

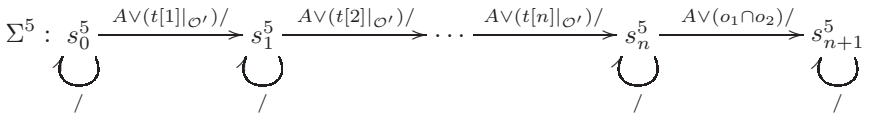
Therefore, Σ has a transition labeled with $/$ in all states s. Along with determinism and the condition that Σ has no infinite sequence of input-less transitions, this implies condition (1).

Property StateDeterm: Consider $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$ with $i_1 \bowtie i_2$. By applying Theorem 1, we obtain $o_1 \bowtie o_2$. To prove the remaining property, we use the same technique as for property (1).

We denote with Δ the symmetric difference operator, on sets and reactions (*e.g.* $i_1 \Delta i_2 = (i_1 \setminus i_2) \vee (i_2 \setminus i_1)$).

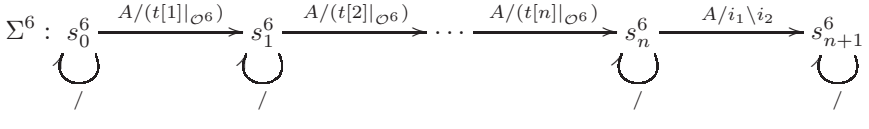
Let $\mathcal{I}' = \mathcal{I} \backslash supp(i_1 \Delta i_2)$ and $\mathcal{O}' = \mathcal{O} \backslash supp(o_1 \Delta o_2)$. Let $\Sigma^i = (\mathcal{I}^i, \mathcal{O}^i, \mathcal{S}^i, s_0^i, \rightarrow^i)$, $i = 4, 5$ be defined by $\mathcal{I}^4 = \{A\}$, $\mathcal{O}^4 = \mathcal{I}'$, $\mathcal{I}^5 = \{A\} \cup \mathcal{O}'$, $\mathcal{O}^5 = \emptyset$, with $A \notin \mathcal{I} \cup \mathcal{O}$, $\mathcal{D}_A = \mathcal{D}_\top$, and transition relations defined by:

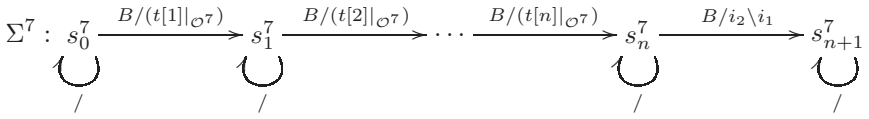


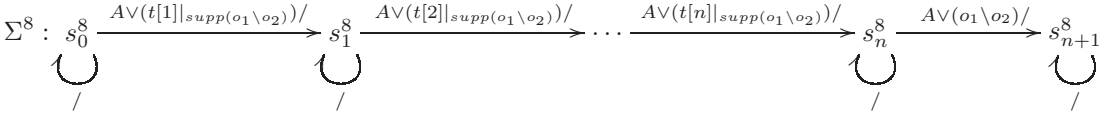


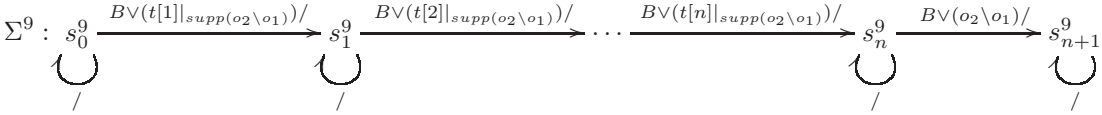
By hypothesis, the composed system $\Sigma^4 \,||\, \Sigma \,||\, \Sigma^5$ must satisfy Properties StreamFunc and StateDeterm. This means that from the reachable states $(s^4_{n+1}, s_1, s^5_{n+1})$ and $(s^4_{n+1}, s_2, s^5_{n+1})$ execution converges. But from the construction of Σ^4 and Σ^5 convergence in the Σ component can only include signals from $\mathcal{I}'$ and $\mathcal{O}'$, so that we can assume in Theorem 1 that $supp(t_i) \subseteq supp(i_1 \Delta i_2) \cup supp(o_1 \Delta o_2)$, $i = 1, 2$.

Let $\Sigma^i = (\mathcal{I}^i, \mathcal{O}^i, \mathcal{S}^i, s^i_0, \rightarrow^i)$, $i = 6, 7, 8, 9$ be defined by $\mathcal{I}^6 = \{A\}$, $\mathcal{O}^6 = supp(i_1 \setminus i_2)$, $\mathcal{I}^7 = \{B\}$, $\mathcal{O}^7 = supp(i_2 \setminus i_1)$, $\mathcal{I}^8 = \{A\} \cup supp(o_1 \setminus o_2)$, $\mathcal{O}^8 = \emptyset$, $\mathcal{I}^9 = \{B\} \cup supp(o_2 \setminus o_1)$, $\mathcal{O}^9 = \emptyset$, with $A, B \notin \mathcal{I} \cup \mathcal{O}$, $\mathcal{D}_A = \mathcal{D}_B \mathcal{D}_\top$, and transition relations defined by:









By hypothesis, the composed system $(\Sigma_6 \,||\, \Sigma_7) \,||\, \Sigma \,||\, (\Sigma_8 \,||\, \Sigma_9)$ satisfies Properties StreamFunc and StateDeterm. But, from the previous property we deduce that the only transition in the confluence of component Σ without label / has label $i_2 \setminus i_1 / o_2 \setminus o_1$ or $i_1 \setminus i_2 / o_1 \setminus o_2$. This proves our result. ◇

To prove the limit result of Theorem 4, we prove the following theorem:

Theorem 6 *Consider $\Sigma = (\mathcal{I}, \mathcal{O}, \mathcal{S}, s_0, \rightarrow)$ a synchronous module of Comp, $s \in RSS(\Sigma)$, and $\chi_\mathcal{I} \in \prod_{S \in \mathcal{I}} \mathcal{D}_S{}^\omega$. Consider $t \in Traces_\Sigma(s)$ a maximal* fair *execution of $[\Sigma]$ under input $\chi_\mathcal{I}$. Then:*

A. *For all finite history $\chi'_\mathcal{I} \preceq \chi_\mathcal{I}$, there exists $n \geq 1$ such that $SF_{\Sigma,s}(\chi'_\mathcal{I}) \preceq \delta(t[1..n])$.*

B. *For all $n \geq 1$, there exists a finite history $\chi''_\mathcal{I} \preceq \chi_\mathcal{I}$ such that $\delta(t[1..n]) \preceq SF_{\Sigma,s}(\chi''_\mathcal{I})$.*

Proof sketch: Point (B) is simply proved by taking $\chi''_\mathcal{I} = \delta(t[1..n]) \mid_\mathcal{I}$.

We focus now on proving (A). The case of finite input traces being covered by previous results, we assume here that $\chi_\mathcal{I} \in (\prod_{S \in \mathcal{I}} \mathcal{D}_S{}^\omega) \setminus (\prod_{S \in \mathcal{I}} \mathcal{D}_S{}^*)$.

Without losing generality, we shall assume that t is stuttering-free. The case of finite input histories and traces being covered by previous theorems, we can assume throughout this proof that both t and $\chi_{\mathcal{I}}$ are infinite.

Let then $\mathcal{I}' \subseteq \mathcal{I}$ be the set of all S such that $\chi_{\mathcal{I}}(S)$ is infinite. We can then find n_1 such that $supp(t[n]) \subseteq \mathcal{I}'$ for all $n \geq n_1$.

Given that Σ is finite state and t infinite, there exists $\mathcal{S}' \subseteq \mathcal{S}$ that are traversed infinitely many times by t. We can also find n_2 such that $s_0.t[1..n] \in \mathcal{S}'$ for all $n \geq n_2$.

Assume (A) is not true, and let $\chi'_{\mathcal{I}} \preceq \chi_{\mathcal{I}}$ such that $SF_{\Sigma,s}(\chi'_{\mathcal{I}}) \not\preceq \delta(t)$. Let then $u \in [\Sigma](\chi'_{\mathcal{I}}, s_0)$. We denote with $t_n = t[1..n]$ and $u_k = u[1..k]$.

Since $\delta(u) = SF_{\Sigma,s}(\chi'_{\mathcal{I}}) \not\preceq \delta(t)$, there exists m such that $\delta(u_m) \preceq \delta(t)$ and $\delta(u_{m+1}) \not\preceq \delta(t)$. Given that $\delta(u_m) \preceq \delta(t)$, there exists n_3 such that $\delta(u_m) \preceq \delta(t_{n_3})$.

Let $n_0 = max(n_1, n_2, n_3)$.

By applying Theorem 5(B) in state s_0 for traces u_m and t_n for some $n \geq n_0$, we obtain $t'_n \in Traces_{\Sigma}(s_0.t_n)$ and $u'_n \in Traces_{\Sigma}(s_0.u_m)$ such that:

$$s_0.t_n.t'_n = s_0.u_m.u'_m \tag{.1}$$

$$\delta(t_n.t'_n) = \delta(u_m.u'_m) \tag{.2}$$

$$\delta(t_n.t'_n) \mid_{\mathcal{I}} = \delta(t_n) \mid_{\mathcal{I}} \vee \delta(u_m) \mid_{\mathcal{I}} \tag{.3}$$

We denote $s_n = s_0.t_n.t'_n$. Given that $\Sigma \in$ *Comp* we can apply Property Stuttering and assume that s_n is chosen such that $s_n = \overline{s_n}$.

Given that $\delta(u_m) \preceq \delta(t)$, we deduce from Equation .3 that $\delta(t'_n) \mid_{\mathcal{I}} = \epsilon$. Combined with the determinism of Σ and the fact that a transition with void label exists in every state, we deduce that $\delta(t'_n) = \epsilon$.

By applying Theorem 2, it is easy to determine that $s_{n+1} = \overline{s_n.t[n+1]}$ for all $n \geq n_0$. This means that for all $n \geq n_0$, $u'_{n+1} = u'_n.t[n+1](/)..(/)$. We denote with u' the limit of the increasing sequence of traces $(u'_n)_{n \geq n_0}$.

By applying Theorem 2 along the intermediate states in the execution of u', we obtain that in all the states s_n, $n \geq n_0$ there exists a transition u^n_{m+1} with

$$\delta(u^n_{m+1}) = \delta(u_{m+1}) \setminus (\delta(u_{m+1}) \wedge \delta(u'_n))$$

The sequence $(\delta(u^n_{m+1}))_{n \geq n_0}$ is a decreasing one, meaning that it eventually stabilizes. From the hypothesis that (A) is not true, it does not stabilize to ϵ, but to $\delta(r)$ for some non-void reaction label r.

From the fairness hypothesis, given that a void label transition is fireable in all the states $s_0.t_n$, $n \geq n_0$, and given that there is only a finite set of such states, at some point the void label transition is taken in the execution of t. This means that there exists $n' \geq n_0$ such that $s_0.t_n = s_n$, which implies that the remainder of t coincides with the remainder of u'.

This means that the reaction labeled with $r \neq \perp$ is fireable in all the states of t from some point on without it being taken, which contradicts our fairness assumption. Therefore (A) is true. $\diamond$

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

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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