From multi-clock constraints to multi-rate GALS executives

Dumitru Potop-Butucaru, Yves Sorel, Robert de Simone

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From multi-clock constraints to multi-rate GALS executives

Dumitru Potop-Butucaru — Robert de Simone — Yves Sorel

N° ????

Novembre 2006

Thème COM

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

From multi-clock constraints to multi-rate GALS executives

Dumitru Potop-Butucaru , Robert de Simone , Yves Sorel

Thème COM — Systèmes communicants
Projet AOSTE

Rapport de recherche n° ???? — Novembre 2006 — 25 pages

Abstract: We define a method for synthesizing the asynchronous executives that are driving the synchronous modules of a globally asynchronous, locally synchronous (GALS) system. The technique takes as input high-level synchronization constraints, under the form of multi-clock modular synchronous reactive (S/R) specifications. For each synchronous module, our technique produces a multi-rate executive that drives the communication and clock of the component using a mixed static/dynamic scheduling policy. The resulting GALS system is predictable and functionally correct with respect to the initial synchronous specification. The approach is based on the theory of weakly endochronous (WE) systems, and on a notion of atomic reaction which allow us to exploit the concurrency of the specification to improve the communication efficiency of the executives.

Key-words: Synchronous, Asynchronous, GALS, Multi-clock, Multi-rate, Weak endochrony, Automatic distribution

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

Synthèse d'exécutifs GALS à partir de contraintes multi-horloges

Résumé : Nous définissons une méthode pour la synthèse des exécutifs asynchrones qui contrôlent l'exécution des modules synchrones d'un système globalement asynchrone, localement synchrone (GALS). Notre technique prend en entrée des contraintes de synchronisation de haut niveau, sous la forme de spécifications synchrones réactives modulaires multi-horloges. Pour chaque module synchrone, notre technique produit un exécutif multi-rythmes qui contrôle les entrées, les sorties, et l'horloge du module en utilisant une politique d'ordonnancement mixte statique/dynamique. Le système GALS résultant est prédictible, correct et complet par rapport à la spécification synchrone d'entrée. L'approche est fondée sur la théorie des systèmes faiblement endochrones (weakly endochronous systems), qui nous permet d'exploiter la concurrence de la spécification pour améliorer l'efficacité des exécutifs.

Mots-clés : Synchrone, Asynchrone, GALS, Multi-horloges, Multi-rythme, Endochronie faible, Répartition automatique

1 Introduction

Development techniques based on synchronous formalisms are today common in various fields, such as digital circuit design or (safety-critical) embedded software development [3]. Synchronous specifications are used in these fields to model concurrent reactive systems. The synchronous model allows the explicit representation of the reaction to signal absence, and thus supports a notion of *deterministic concurrency* which facilitates functional modelling and analysis. Implementing synchronous specifications is often hard. This is mainly due to the global synchronization mechanisms of the model, which need to be preserved, at least in part, to ensure that absence information is not lost when it can influence the behavior of the system.

In this paper, we address the synthesis of the asynchronous executives that are driving the synchronous modules of a GALS system. The topic draws much attention today, when distributed embedded systems and complex systems-on-chip (SoC) are commonplace in the industry. Our synthesis technique is mathematically founded on the theory of *weakly endochronous systems*, due to Potop, Caillaud, and Benveniste [16]. Weak endochrony gives criteria establishing that a synchronous presentation hides a behavior where the absence information is never needed, so that the synchronous specification can safely be executed in an asynchronous environment, with predictible results. Weak endochrony extends to a synchronous framework the Mazurkiewicz traces [12]. Thus, the current paper bridges between the fields of synchronous language compilation and classical concurrency theory.

The implementation technique is defined as an *assisted synthesis technique* that takes as input high-level synchronization constraints structured in a multi-clock synchronous specification. The transformation is performed in two steps. The first one checks whether the specifications of the synchronous modules are weakly endochronous. If they are not, intuitive diagnostics allow the user to incrementally improve the signalling protocols. When the specification is weakly endochronous, the second translation step automatically synthesizes the GALS executives.

Outline. The paper is organized as follows: Section 2 intuitively presents our problem, previous work, and the desired solution. Section 3 defines the formalism that will support our presentation. Section 4 is on weak endochrony: the original theory of [16] and algorithms to determine if a specification is WE. Section 5 explains how the asynchronous executives are generated for WE components. Section 6 gives criteria ensuring the correct functioning of the resulting GALS system, and Section 7 concludes.

2 Overview

Our goal is the construction of correct and predictable executives for GALS systems. In this paper, we consider this problem in an untimed setting. Taking into account or ensuring the satisfaction of real-time constraints [4] shall be the subject of future work.

2.1 Distributed implementation

Our basic model of distributed system is similar to that of Kahn Process Networks (KPN) [10]. A system is formed of computing *components*. The components communicate through *message passing* along *lossless order-preserving asynchronous lines*. For simplicity, our examples only use simple point-to-point communication lines—*asynchronous FIFOs*.

Reading or writing a message on a communication line is blocking and non-interruptible. Blocking reads are part of the basic KPN model. Blocking writes are needed to ensure flow regulation in the absence of timing or synchronization information on the environment. With blocking writes, we can use real-life bounded-memory communication lines instead of the unbounded FIFOs of the basic Kahn model.

2.2 Modular synchronous specification

To synthesize our executives, we start with the set of synchronous modules of the GALS system: *off-the-shelf synchronous IPs* or compiled synchronous code, under the form of software *reaction functions*. Following the terminology of [5], we call these implementation modules *pearls*.

Each pearl is accompanied by a high-level specification of its synchronization properties. These specifications need not be functionally complete w.r.t. to the pearls. They should contain enough synchronization information allowing the construction of the asynchronous executives controlling them. Irrelevant information, such as complex data operations, can, and should be abstracted away, to facilitate analysis.

The input of our synthesis technique is the modular synchronous specification obtained by putting in parallel the high-level descriptions of all the pearls. In the formalism which will support our presentation (a small sub-set of the Signal language [8]), this means that the specification is the parallel composition of a number of *processes*.

We use a small, intuitive example to present our problem, the desired result, and the main implementation issues. The example, pictured in Fig. 1, is a simple reconfigurable adder, where two independent single-word ALUs can be used either independently, or synchronized to form a double-word ALU. The choice between synchronized and non-synchonized mode is done using the `SYNC` signal. To simplify the figure, we compacted both data inputs of each adder under a single signal name (`I1` and `I2`, respectively).

The carry between the two adders is propagated through the `C` wire whenever `SYNC` is present. The two single-word ALUs are the processes of our specification.

As we shall see later in this paper, the high-level specification of our example can be functionally incomplete. It can safely abstract away the actual behavior of the adders, and the integer data.

2.2.1 Synchronous reactions

Like any synchronous formalism, we follow a discrete model of time, where executions are sequences of *reactions*, indexed by a *global clock*. Table 1 gives a possible execution of

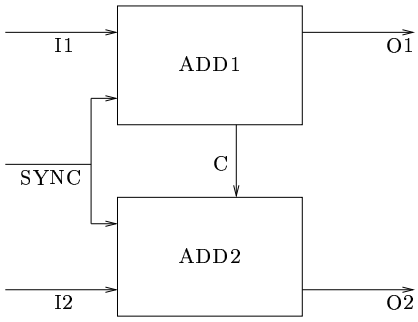

Figure 1: Configurable adder: global dataflow

Clock	1	2	3	4	5	6	7
I1	(1,2)	$\perp$	(9,9)	(9,9)	$\perp$	(2,5)	$\perp$
O1	3	$\perp$	8	8	$\perp$	7	$\perp$
SYNC	$\perp$	$\perp$	$\top$	$\perp$	$\perp$	$\top$	$\perp$
C	$\perp$	$\perp$	1	$\perp$	$\perp$	0	$\perp$
I2	$\perp$	$\perp$	(0,0)	(0,0)	$\perp$	(1,4)	(2,3)
O2	$\perp$	$\perp$	1	0	$\perp$	5	5

Table 1: Sample synchronous run of the example

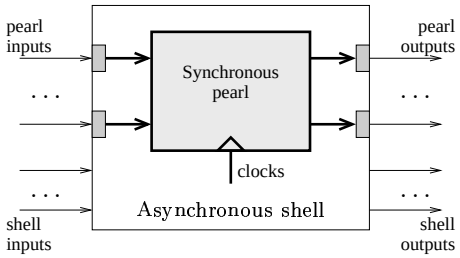

Figure 2: Desired component structure

our example. A reaction is a valuation of the input, output, and internal *signals* of the process. We shall denote with $\mathcal{V}$ the finite set of signals of a process. In our example, $\mathcal{V} = \{I1, I2, SYNC, O1, O2, C\}$.

All signals are typed. We denote with $\mathcal{D}_{\mathtt{S}}$ 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 process compute. We will say that a signal is *present* in a reaction when it has a value in $\mathcal{D}_{\mathtt{S}}$. Otherwise, we say that it is *absent*. Absence is simply represented with value $\perp$, which is appended to all domains $\mathcal{D}_{\mathtt{S}}^{\perp} = \mathcal{D}_{\mathtt{S}} \cup \{\perp\}$. Formally, a reaction of the process is a partial valuation of its signals. We denote with $\mathcal{R}$ the set of all such valuations. The *support* of a reaction r, denoted $supp(r)$, is the set of present signals. For instance, the support of reaction 4 in Table 1 is $\{I1, I2, O1, O2\}$.

In many cases we are only interested in the presence or absence of a signal, because it transmits no data, or because we are only interested in synchronization aspects. To represent such signals, the Signal language uses a dedicated `event` type of domain $\mathcal{D}_{\mathtt{event}} = \{\top\}$. In our example, `SYNC` has type `event`.

To represent reactions, we use a *set-like convention* and omit signals with value $\perp$. In Fig. 1, the signal set is $\{SYNC : event, O1, O2 : integer, I1, I2 : integer\_pair\}$. In Table 1, reaction 4 is denoted $(I1^{(9,9)}, O1^{8}, I2^{(0,0)}, O2^{0})$. The *stuttering reaction* assigning $\perp$ to all signals is denoted $\perp$. In Table 1, reaction 5 is a stuttering reaction.

2.3 Structure of a computing component

To interface between the asynchronous communication lines and the synchronous pearls, we structure our computing components as pictured in Fig. 2.

We are interested here in the asynchronous executive, also called *shell*, which controls the asynchronous I/O and drives the synchronous pearl of the component. The shell is basically an asynchronous automaton[1]. It takes as input the asynchronous FIFOs transmitting the inputs of the pearl, and it can also take additional control inputs added by the synthesis process to ensure correct synchronization. When enough input signals are present, the shell

[1] But its implementation can be synchronous, like in [5], or simply sequential.

I1	(1,2)	(9,9)	(9,9)	(2,5)
O1	3	8	8	7
SYNC		$\top$	$\top$	
C		1	0	
I2	(0,0)	(0,0)	(1,4)	(2,3)
O2	1	0	5	5

Table 2: Corresponding asynchronous run. Correctly reconstructing synchronous inputs from the asynchronous ones is impossible.

triggers a synchronous reaction of the pearl. To do this, the shell provides the pearl with the needed inputs, and marks all other pearl inputs as *absent* ($\perp$) even if values have been received for them. Then, it triggers the computation of the reaction. In software, this amounts to calling the reaction function. In hardware, the pearl clock is enabled. Once the pearl completes its computation: In hardware, the pearl clock is disabled; In software, the state update protocol is performed. The outputs computed by the pearl are transmitted onto the output FIFOs. The input values used in the reaction are acknowledged as read, so that new ones can be accepted from the corresponding FIFOs.

2.4 The synthesis problem

We need to synthesize the (hardware or software) shells that will drive the execution of the the pearls on the components of the architecture.

The main task of the shell is to construct in a consistent way inputs for the synchronous pearl. The input of a reaction must assign a value, or declare absent, every input of the pearl. With this hypothesis, code generation for the pearl is highly simplified[2], for it does not have to infer or enforce input presence or absence, nor buffer inputs or outputs to match the asynchronous transmission protocols.

Reconstructing reactions from asynchronous messages must be done in a deterministic fashion, regardless of the message arrival order. This is not always possible. Assume, like in Table 2, that we consider the inputs of Table 1 without the synchronization information. The module `ADD1` will then receive the first value $(1, 2)$ on the input channel `I1` and $\top$ on `SYNC`. Depending on the arrival order, which cannot be precised, any of the reactions $(I1^{(1,2)}, O1^3, SYNC^{\top}, C^0)$ or $(I1^{(1,2)}, O1^3)$ can be triggered, leading to divergent computations. The problem is that the two reactions are not independent (they do not commute), yet the choice between them cannot be done over the value of a message on a given channel, so a shell has no means of choosing between them. A similar problem occurs in `ADD2`.

Our approach is to transform the synchronous specification in such a way as to make reaction reconstruction deterministic upto commutation of independent reactions. One such implementation of our example is presented in Fig. 3, with an execution trace in Table 4.

[2]The code can be *exochronous*, in the sense of [14].

Transformation is done by adding communication lines. To discriminate between the non-idependent transitions of `ADD1`, we introduce the Boolean signal `SYNC1`. Value 1 on `SYNC1` indicates that the two adders synchronize, so that `SYNC` and `C` are present. Value 0 indicates an independent computation on `ADD1`. Symmetrically, `SYNC2` is added to `ADD2`. Note that signal `SYNC` becomes useless, as it corresponds to choices over `SYNC1` and `SYNC1`. Therefore, we can delete it from the dataflow scheme. Note, in Table 4, that the two processes can be now safely run on separate clocks in the GALS implementation.

2.5 Previous solutions

A particular class of solutions to our problem has already been thoroughly studied: The case where every pearl reads all inputs and writes all outputs at each reaction (none can be absent). We shall call this the mono-clock case, for reasons that will become clear in Section 3. In this case, implementations have been given in both software and hardware:

- In SynDEx [7], mono-clock synchronous specifications are the input of optimized multi-processor scheduling algorithms. The computations of the pearls are statically scheduled on each processor.
- In Latency-Insensitive Design (LID) [5], the model serves as basis for the synthesis of technology-independent on-chip communication protocols.

Putting our example in mono-clock form would consist in explicitly transmitting over the asynchronous lines the absence of signals, *i.e.*, transmitting all the symbols in Table 1. This may be inefficient when we need to minimize communication, or when we want to allow multi-rate computation.

The first extension towards multi-clock pearls were the *endochronous systems* [2, 8] and the related hardware *generalized latency-insensitive systems* [17]. Here, we assume that the presence and absence of all signals can be incrementally inferred starting from the state and from signals that are always present. Table 3 presents a run of an endochronous system obtained by transforming the `SYNC` signal of our example into one that carries values from 0 to 3: 0 for `ADD1` executing alone, 1 for `ADD2` executing alone, 2 for both adders executing without synchronization (`C` absent), and 3 for the synchronized execution of the two adders (`C` present). Note that the value of `SYNC` determines the presence/abence of all signals. The compilation of the Signal language is currently founded on a version of endochrony [1].

The problem with endochrony is that it does not allow concurrency inside synchronous modules. The reaction reconstruction process is fully deterministic, and the presence of all signals is synchronized w.r.t. some base signal(s) in a hierarchic fashion. If we refer to our example, an endochronous component cannot allow concurrency between independent computations of the two adders. Thus, we cannot allow implementations where both `ADD1` and `ADD2` are implemented on the same pearl without removing all independence between them through added signalling, even though this is not functionally necessary. This leads to over-synchronizing the implementation, and to potential inefficiency. As a side-effect,

Clock	1	2	3	4	5
I1	(1,2)	(9,9)	(9,9)	(2,5)	$\perp$
O1	3	8	8	7	$\perp$
SYNC	0	3	2	3	1
C	$\perp$	1	$\perp$	0	$\perp$
I2	$\perp$	(0,0)	(0,0)	(1,4)	(2,3)
O2	$\perp$	1	0	5	5

Table 3: Endochronous solution

Clock1	1	2	3		4	
I1	(1,2)	(9,9)	(9,9)		(2,5)	
O1	3	8	8		7	
SYNC1	0	1	0		1	
C		1			0	
SYNC2		1		0	1	0
I2		(0,0)		(0,0)	(1,4)	(2,3)
O2		1		0	5	5
Clock2		1		2	3	4

Table 4: Weakly endochronous solution

endochrony is not compositional, which makes the definition of incremental development processes difficult.

2.6 Our approach

In this paper, we explain how to allow both synchronized and non-synchronized (independent) computations to be realized by a single component. In this approach, the user can decide at pearl synthesis time (maybe through an exploration process) how to group the processes onto the distributed components, while knowing that efficient shells are automatically generated. For instance, the two processes of our example can be implemented by separate pearls, or by a single one.

The main contribution is given in Section 4. We explain there how to determine whether a process is weakly endochronous. This is based on determining a minimal set of reactions that generate all other reactions. The process is weakly endochronous when all generators satisfy an *atomicity hypothesis* – they are either asynchronously distinguishable, or non-interferent. When a process is not weakly endochronous, we need to constrain it further by eliminating non-confluent choices that are not determined by a signal value (not by absence). Given the complexity of this step, which adds supplementary communication lines, we do not fully

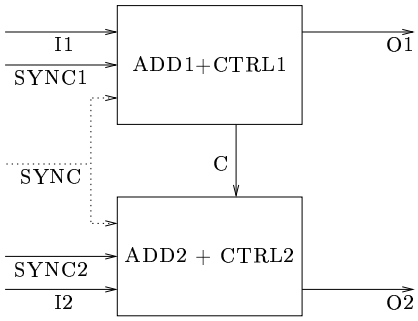

Figure 3: Weakly endochronous solution

automate it. Istead of providing solutions that might not match the global architecture of the system, our analysis algorithm simply points out through intuitive error messages why the system is not weakly endochronous. Once the assisted transformation step is completed, the asynchronous shells are automatically generated, as explained in Section 5.

2.6.1 Stateless abstraction

To keep the presentation simple, we present here a simple version of the technique, which does not take into account the process state. The approach can be extended to deal with specifications where each process has a state.

3 Synchronous Dataflow in Signal

We use a small sub-set of the high-level synchronous language Signal to formally present our technique. The Signal multi-clock constraint language allows a simple representation of the synchronization constraints we use. However, the results can be simply applied to any synchronous dataflow specification accompanied by synchronization constraints. In particular, SynDEx [7] and the latency-insensitive design [5] could be extended using our results.

3.1 Process structure

A Signal process is a *hierarchical dataflow specification*. Fig. 4 gives the Signal process corresponding to the configurable adder of Fig. 1.

A process is formed of a header defining its name, an interface specification, a dataflow specification, and local declaration section. In our example, the top process is named `EXAMPLE`. Its interface defines 3 input signals (`SYNC`, `I1`, and `I2`), identified with "?", and 2 output signals (`O1` and `O2`), identified with "!". Given a process P, we denote with $\mathcal{I}(P)$

```
process EXAMPLE =
(? event SYNC ; event I1,I2 ;
 ! event O1,O2 ; )
(| (O1,C) := ADD1 (SYNC,I1)
 | O2 := ADD2 (SYNC,I2,C) |)
where boolean C ;

  process ADD1 =
  (? event SYNC ; event I1 ;
   ! event O1 ; event C ; )
  (| I1 ^= O1 | SYNC ^< I1 | C ^= SYNC |) ;

  process ADD2 =
  (? event SYNC ; event I2 ; event C ;
   ! event O2 ; )
  (| I2 ^= O2 | SYNC ^< I2 | C ^=SYNC |) ;
end ;
```

Figure 4: The Signal process of the configurable adder in Fig. 1

its set of input signals, with $\mathcal{O}(P)$ its set of output signals, and with $\mathcal{V}(P)$ the set of all its signals. By definition, $\mathcal{I}(P) \cap \mathcal{O}(P) = \emptyset$ and $\mathcal{I}(P) \cup \mathcal{O}(P) \subseteq \mathcal{V}(P)$.

The dataflow specification of `EXAMPLE` consists of two equations, which define the interconnections between `ADD1`, `ADD2`, and the environment. The local definition section defines the internal signal `C`, and the processes `ADD1` and `ADD2`.

Our synthesis technique will only refer directly to the first two levels of the process hierarchy. The unique top-level process is the specification, and it only contains the global dataflow. Its children are the specifications of the synchronous modules, which can contain more complex behavioral Signal definitions. Lower-level processes cannot be distinguished from the specification of the module that contains them.

3.1.1 Finite abstraction

As explained in section Section 2.2, complex data operations that are irrelevant to the synthesis of the shells should be abstracted away in the synchronous specification. For instance, we model the integer inputs and outputs of the two adders with `event` signals. Data abstraction is mandatory for all infinite types (e.g., `integer`, `float`), to make decidable the analysis of the weak endochrony. We say that the synchronous specification is a *finite stateless abstraction* of the corresponding pearl behavior.

3.2 Dataflow specification

The dataflow specification of a process is formed of *equations* defining *constraints* between the signals of the process. Any reaction satisfying all the equations of a process P is a valid reaction of P. We denote with $\mathcal{R}(P)$ the set of all the reactions of P. Some of the constraints (such as assignment) are often used to represent the actual flow of data in the implementation. We do this only in the top-level process of a specification. In sub-processes, the assignment operator `:=` is manipulated according to its basic semantics: an equality constraint with no causality information.

The use of a constraint language allows us to easily manipulate functionally incomplete specifications. For instance, we do not need to represent the full behavior of the adders `ADD1` and `ADD2`. We only specify the synchronization information that is necessary for the synthesis of the communication protocols.

3.2.1 Clocks. Clock Constraints

The *clock* of a signal `S` is another signal, denoted `^S`, of type `event`, which is present whenever `S` is present. Clock signals are used to specify *clock constraints*.

The most common clock constraints are *identity*, *inclusion*, and *exclusion*. Line 11 of Fig. 4, which gives the constraints of `ADD1`, illustrates clock equality and inclusion. The equation "`I1^=O1`" specifies that signal `I1` is present in a reaction *iff* `O1` is present. In other terms, whenever inputs arrive, the adder produces an output. The next equation requires that `I1` is present in reactions where `SYNC` is present. Otherwise said, `^SYNC` is included in `^I1`. The last equation states that the carry value `C` is emitted by `ADD1` whenever `SYNC` is present.

The definition of `ADD2` is similar. The difference is that the carry signal `C` is here an input, and not an output like in `ADD1`.

The equation "`S1^#S2`" specifies the exclusion between the clocks of the signals `S1` and `S2`, *i.e.*, the fact that both signals cannot be present in a reaction.

3.2.2 Stateless Signal primitive language

The following statements are the primitives of the Signal language sub-set we consider. The delay primitive of the full language is missing, because in this paper we only consider stateless specifications.

The assignment equation "`X:=f(Y1,...,Yn)`" states that all the signals have the same clock, and that the specified equality relation holds at each instant where the signals are present. Equation "`X:=Y`" is a particular case of assignment. It specifies the identity of `X` and `Y`. Signal `Y` can also be replaced with a dataflow expression built using the operators defined below.

The operator `when` performs conditional downsampling. The signal "`X when C`" is equal to `X` whenever the `boolean` signal `C` is present with value `true`. Otherwise, it is $\perp$. The shortcut for "`^C when C`" is "`when C`". For instance, in Fig. 5, "`when SYNC1=1`" is a signal of type `event` that is present when signal `SYNC1` is present with value 1.

```
process EXAMPLE_RESULT =
(? boolean SYNC1,SYNC2; event I1,I2 ;
 ! event O1,O2 ; )
(| (O1,C) := CTRL1 (SYNC1,I1)
 | O2 := CTRL2 (SYNC2,I2,C) |)
where boolean C ;

  process CTRL1 =
  (? boolean SYNC1 ; event I1 ;
   ! event O1 ; event C ; )
  (| SYNC1 ^= I1 ^= O1 | C ^= when SYNC1=1
   | (O1,C) := ADD1(when SYNC1=1,I1) |)
  where process ADD1 = ... end ;

  process CTRL2 =
  (? boolean SYNC2 ; event I2 ; event C ;
   ! event O2 ; )
  (| SYNC2 ^= I2 ^= O2 | C ^= when SYNC2=1
   | O2 := ADD2(when SYNC2=1,I2,C) |)
  where process ADD2 = ... end ;
end ;
```

Figure 5: Weakly endochronous process corresponding to Fig. 3. Processes `ADD1` and `ADD2` are copied from Fig. 4.

The operator `default` merges two signals of the same type, giving priority to the first. The signal "`X default Y`" is present whenever one of `X` or `Y` is present. It is equal to `X` whenever `X` is present, and is equal to `Y` otherwise.

A Signal process can instantiate other Signal processes inside its dataflow. In Fig. 4 `EXAMPLE` instantiates `ADD1` and `ADD2`. In Fig. 5 `CTRL1` instantiates `ADD1`.

3.3 Notations

A synchronous trace of a process P is any finite succession of reactions of P. The set of traces of P is $\mathcal{T}(P) = \mathcal{R}(P)^*$.

On the values of any signal S, we introduce a partial order, given by $\perp \leq v$, for all $v \in \mathcal{D}_S$. The order on signal values induces a product partial order $\leq$ on reactions. Note that $r_1 \leq r_2$ if and only if $supp(r_1) \subseteq supp(r_2)$ and $r_1(v) = r_2(v)$ for all $v \in supp(r_1)$.

We say of two reactions r_1 and r_2 that they are *non-contradictory*, written $r_1 \bowtie r_2$, if $r_1(v) = r_2(v)$ for all $v \in supp(r_1) \cap supp(r_2)$. Otherwise, we say that the reactions are *contradictory*, written $r_1 \not\bowtie r_2$. Given a set of reactions K, we shall say that it is non-contradictory, denoted $\bowtie K$ if any two reactions of K are non-contradictory. In this case,

we can define $\vee K = \bigvee_{r \in K} r$. Two reactions r_1 and r_2 are *non-interferent* if $supp(r_1) \cap supp(r_2) = \emptyset$.

On non-contradictory reactions, we define the union $\vee$ and intersection $\wedge$ operators as the least upper bound and greatest lower bound induced by $\leq$. We also define the difference $r_1 \setminus r_2$, which has support $supp(r_1) \setminus supp(r_2)$ and equals r_1 on its support.

4 Weak endochrony

4.1 Basic theory

The theory of *weakly endochronous (WE) systems* [16], gives criteria establishing that a synchronous presentation hides a behavior that is fundamentally asynchronous and deterministic. Absence information is not needed, which guarantees the deterministic implementability of the synchronous specification in an asynchronous environment. Weak endochrony extends to a synchronous framework the Mazurkiewicz traces [12].

Weak endochrony is defined in an automata-theoretic framework. We simplify it here according to our stateless abstraction:

Definition 1 (stateless weak endochrony) *We say that process P is weakly endochronous if for all $r_1, r_2 \in \mathcal{R}(P)$*

$$r_1 \bowtie r_1 \Rightarrow r_1 \vee r_2, r_1 \setminus r_2, r_1 \wedge r_2, r_2 \setminus r_1 \in \mathcal{R}(P)$$

We already saw, in Section 2.4, that running a synchronous process in an asynchronous environment may be a non-deterministic process. Weak endochrony gives very general sufficient conditions ensuring that the execution of the process always gives the same result, regardless of the asynchronous input arrival order.

The intuition behind weak endochrony is that we are looking for systems where (1) all causality is implied by the sequencing of messages on communication channels, and (2) all choices are visible as choices over the value (and not present/absent status) of some message. As explained in [15], the axioms of weak endochrony can be traced down to the fundamental result of Keller [11] on the deterministic operation of a system in an asynchronous environment. From a different perspective, WE systems are synchronous Kahn processes, with similar determinism results.

4.1.1 Atoms

From our point of view oriented towards code generation, the most interesting consequence of these axioms is that any behavior of a WE system can be decomposed into *atomic transitions*, or *atoms*. Formally, the set of atomic reactions of P, denoted $Atoms(P)$ is the set of the smallest[3] reactions of $\mathcal{R}(P)$ different from $\perp$.

[3] In the sense of $\leq$.

Atomic transitions have very nice properties. Two atoms a_1 and a_2 are either non-interferent (of disjoint support, cf. Section 3.3), or contradictory. Furthermore, any transition of the system can be uniquely decomposed into a set of non-interferent atoms.

Theorem 1 (Generation) *Let P be a weakly endochronous stateless process, and $r \in \mathcal{R}(P)$. Then:*

$$r = \bigvee_{a \in Atoms(P), a \leq r} a$$

4.2 Checking weak endochrony

In this section we explain how to check the weak endochrony of the process corresponding to a synchronous module of the GALS system. The analysis is based on the identification of a set of reactions which generate by union all other reactions. The generator set is computed incrementally from the equations of the specification. The process is WE when the generator set has the properties of an atom set. If the process is WE, then the atoms we computed will directly serve in Section 5 to generate the asynchronous shell.

4.2.1 Partial reactions

The domain of signal valuations, defined in Section 2.2.1, allows the representation of complete reactions of processes, as well as the manipulation of reactions of weakly endochronous processes, where absence information is not needed. However, they do not allow the representation of absence synchronizations constraints, such as clock exclusion. To represent such constraints, we enrich the domain $\mathcal{D}_S^{\perp}$ of each signal with a new value $\perp\!\!\!\perp$. Thus, $\mathcal{D}_S^{\perp\!\!\!\perp} = \mathcal{D}_S^{\perp} \cup \{\perp\!\!\!\perp\}$, with $\perp \leq \perp\!\!\!\perp$.

A partial reaction r is any valuation of the signals over their extended domains. We denote with $\mathcal{R}^{\perp\!\!\!\perp}$ the set of all such valuations. All the operators of Section 3.3 are extended over $\mathcal{R}^{\perp\!\!\!\perp}$, according to the new domain structure. The support of a partial reaction is the set of signals with value different from $\perp$.

A partial reaction r sets signal S to $\perp\!\!\!\perp$ to represent the fact that values of other signals force S to be absent. For instance, if "`S^#T`" and $r(S) \in \mathcal{D}_S$, we need $r(T) = \perp\!\!\!\perp$. The partial reaction $(S^v, T^{\perp\!\!\!\perp})$ cannot be united (using $\vee$) with other partial reactions where T is present. This is different from $(S^v, T^{\perp})$, which can be composed, for instance, with (T^v).

Any reaction is a partial reaction, and any partial reaction r has an associated reaction $[r]$ which changes all $\perp\!\!\!\perp$ values of r into $\perp$ values. We denote with $\mathcal{R}^{\perp\!\!\!\perp}(P) = \{r \in \mathcal{R}^{\perp\!\!\!\perp} \mid [r] \in \mathcal{R}(P)\}$

4.2.2 Generators

In this section, we define the non-interferent generator set of a process, and we explain how to compute it. Non-interferent generators of the reactions of a process can be compared with the prime implicants of a logic formula – they are reactions of smallest support that generate

all other reactions. The main difference is that our algebraic structure is richer than that of the Boolean logic (it has a domain structure), allowing us to exploit concurrency to preserve fewer generators (exponentially fewer, in certain cases). Also, it must be noted that $\perp$ is always a solution, in our case.

Definition 2 (Generator set) *Let P be a process. A set $\mathcal{C} \subseteq \mathcal{R}^{\perp\!\!\!\perp}(P)$ of partial reactions is a generator set of $\mathcal{R}(P)$ if $\mathcal{R}(P) = \{[\bigvee_{r\in K} r] \mid K \subseteq \mathcal{C} \wedge \bowtie K\}$.*

Definition 3 (Non-interferent generator set) *Let P be a process. A generator set $\mathcal{C}$ is called a non-interferent generator set of P if:*

$$\forall r_1, r_2 \in \mathcal{C} : \left.\begin{array}{c} r_1 \bowtie r_2 \\ r_1 \neq r_2 \end{array}\right\} \Rightarrow supp([r_1]) \cap supp([r_2]) = \emptyset \qquad (\mathcal{NI})$$

Non-interference implies a strong form of minimality for a generator set, also facilitating the manipulation of the generators.

Theorem 2 *Any process has a unique non-interferent generator set. When the process is weakly endochronous, the non-interferent generator set defines the set of atoms of the process.*

This result shall be proved inductively, by construction of the unique non-interferent generator set. We shall denote the non-interferent generator set associated with a set of equations p of a signal process P with $\mathcal{G}(P, p)$. In the notation, process P is needed to specify the signal set over which p specifies constraints.

The following are non-interferent generator sets for the primitives.

$\mathcal{G}(P, \mathtt{X} := \mathtt{Y}\ \mathtt{default}\ \mathtt{Z}) =$

$$\begin{array}{rl} = & \{(X^v, Y^v, Z^w) \mid v \in \mathcal{D}_X, w \in \mathcal{D}_X \cup \{\perp\!\!\!\perp\}\} \cup \\ & \{(X^v, Y^{\perp\!\!\!\perp}, Z^v) \mid v \in \mathcal{D}_X\} \cup \\ & \{(W^v) \mid v \in \mathcal{D}_X, W \in \mathcal{V}(P) \setminus \{X, Y, Z\}\} \end{array}$$

$\mathcal{G}(P, \mathtt{X} := \mathtt{Y}\ \mathtt{when}\ \mathtt{Z}) =$

$$\begin{array}{rl} = & \{(X^v, Y^v, Z^1) \mid v \in \mathcal{D}_X\} \cup \\ & \{(X^{\perp\!\!\!\perp}, Y^v, Z^w) \mid v \in \mathcal{D}_X \cup \{\perp\!\!\!\perp\}, w \in \{\perp\!\!\!\perp, 0\}\} \cup \\ & \{(W^v) \mid v \in \mathcal{D}_X, W \in \mathcal{V}(P) \setminus \{X, Y, Z\}\} \end{array}$$

$\mathcal{G}(P, \mathtt{X} := \mathtt{f}(\mathtt{Y}_1, \ldots, \mathtt{Y}_n)) =$

$$\begin{array}{rl} = & \{(X^{f(v_1,\ldots,v_n)}, {Y_1}^{v_1}, \ldots, {Y_n}^{v_n}) \mid \forall i : v_i \in \mathcal{D}_{Y_i}\} \cup \\ = & \{(X^{\perp\!\!\!\perp}, {Y_1}^{\perp\!\!\!\perp}, \ldots, {Y_n}^{\perp\!\!\!\perp})\} \cup \\ & \{(W^v) \mid v \in \mathcal{D}_X, W \in \mathcal{V}(P) \setminus \{X, Y_i \mid 1 \leq i \leq n\}\} \end{array}$$

We also give here generators sets of other, non-primitive equations:

$$\begin{aligned}
\mathcal{G}(P, \mathtt{X} := \mathtt{Y}) &= \{(X^v, Y^v) \mid v \in \mathcal{D}_X \cup \{\perp\!\!\!\perp\}\} \cup \\
&\quad \{(W^v) \mid v \in \mathcal{D}_X, W \notin \{X, Y\}\} \\
\mathcal{G}(P, \mathtt{X}\hat{}=\mathtt{Y}) &= \{(X^v, Y^w) \mid v, w \in \mathcal{D}_X\} \cup \{(X^{\perp\!\!\!\perp}, Y^{\perp\!\!\!\perp})\} \cup \\
&\quad \{(W^v) \mid v \in \mathcal{D}_X, W \notin \{X, Y\}\} \\
\mathcal{G}(P, \mathtt{X}\hat{}\#\mathtt{Y}) &= \{(X^v, Y^{\perp\!\!\!\perp}), (X^{\perp\!\!\!\perp}, Y^v) \mid v \in \mathcal{D}_X\} \cup \\
&\quad \{(W^v) \mid v \in \mathcal{D}_X, W \notin \{X, Y\}\} \\
\mathcal{G}(P, \mathtt{X}\hat{}<\mathtt{Y}) &= \{(X^v, Y^w) \mid v \in \mathcal{D}_X \cup \{\perp\!\!\!\perp\}, w \in \mathcal{D}_Y\} \cup \\
&\quad \{(W^v) \mid v \in \mathcal{D}_X, W \notin \{X, Y\}\}
\end{aligned}$$

Theorem 3 (Merging) *Let P be a process and p and q be sets of equations of P. Then:*

1. *If p is a single equation, then $\mathcal{G}(P, p)$ defined above give the non-interferent generator set of p in P.*
2. *Procedure 1 (MergeGeneratorSets) computes $\mathcal{G}(P, p \mid q)$, the unique non-interferent generator set of $p \mid q$ in P.*

Procedure 1 (MergeGeneratorSets) computes in $\mathcal{G}(P, p \mid q)$ all the minimal reactions of $p \mid q$. It explores every minimal combination of compatible atoms in p and q until either a common reaction is found, or no continuation is possible. The search proceeds by constructing two compatible partial reactions, one of p, another of q. The support of the two partial reactions is enlarged, in alternate reactions, only if it is necessary.

Proof of Theorem 3: For a single statement p, $\mathcal{G}(P, p)$ is clearly the non-interferent generator set. We need to prove the induction step.

First, we prove that $\mathcal{G}(P, p \mid q)$ satisfies the non-interference property ($\mathcal{NI}$). Assume $g_1, g_2 \in \mathcal{G}(P, p \mid q)$ such that $g_1 \bowtie g_2$. Then:

$$g_1, g_2 \in \{\bigvee_{r \in K} r \mid K \subseteq \mathcal{G}(P, p) \wedge \bowtie K\}$$

Let $g_1' \in \mathcal{G}(P, p)$ part of the decomposition of g_1, assigning a signal $v \in supp(g_1) \cap supp(g_2)$. Then, there exists $g_2' \in \mathcal{G}(P, p)$, part of the decomposition of g_2, assigning v. Since $g_1 \bowtie g_2$, we have $g_1' \bowtie g_2'$. From the construction, we have $v \in supp([g_1']) \cap supp([g_2']) \neq \emptyset$. Then, by applying the induction hypothesis ($\mathcal{NI}$ on $\mathcal{G}(P, p)$), $g_1' = g_2'$. We can conclude that any generator of g_1 or g_2 touching the intersection of g_1 and g_2 is completely included in the intersection. Therefore, $g_1 \wedge g_2$, $g_1 \setminus g_2$, and $g_2 \setminus g_1$ are partial reactions in p. Similarly, they are partial reactions in q, meaning that they are partial reactions in $p \mid q$. If more than one of them is non-empty, we have a contradiction with the construction process, which results in minimal reactions. When only one set is non-empty, we retrieve the non-interference property.

It remains to prove that $\mathcal{G}(P, p \mid q)$ is a generator set. Any reaction of $p \mid q$ is covered (by the maximal common reactions included in the reaction itself). $\diamond$

Procedure 1 MergeGeneratorSets

Input: $\mathcal{G}(P,p)$, $\mathcal{G}(P,q)$: reaction set
Output: $\mathcal{G}(P,p \mid q)$: reaction set
  $\mathcal{G}_p \leftarrow \emptyset$; $\mathcal{G}'_p \leftarrow \emptyset$;
  for all $g \in \mathcal{G}(P,p)$ **do**
    if $[g] = \perp$ **then** $\mathcal{G}'_p \leftarrow \mathcal{G}'_p \cup \{g\}$
    else $\mathcal{G}_p \leftarrow \mathcal{G}_p \cup \{g\}$
  $\mathcal{G}_q \leftarrow \emptyset$; $\mathcal{G}'_q \leftarrow \emptyset$;
  for all $g \in \mathcal{G}(P,q)$ **do**
    if $[g] = \perp$ **then** $\mathcal{G}'_q \leftarrow \mathcal{G}'_q \cup \{g\}$
    else $\mathcal{G}_q \leftarrow \mathcal{G}_q \cup \{g\}$
  for all $g \in \mathcal{G}_p$ **do**
    while $\exists g' \in \mathcal{G}'_q$ with $g \bowtie g'$, $g' \not\leq g$, and $supp(g) \cap supp(g') \neq \perp$ **do**
      $g \leftarrow g \vee g'$
  for all $g \in \mathcal{G}_q$ **do**
    while $\exists g' \in \mathcal{G}'_p$ with $g \bowtie g'$, $g' \not\leq g$, and $supp(g) \cap supp(g') \neq \perp$ **do**
      $g \leftarrow g \vee g'$
  $\mathcal{G}(P,p \mid q) \leftarrow \mathcal{G}'_q \cup \mathcal{G}'_p$
  for all $g \in \mathcal{G}_p$ **do**
    MergeGenerators(g,$\perp$, $\mathcal{G}_p$,$\mathcal{G}_q$, $\mathcal{G}(P,p \mid q)$)

Procedure 2 MergeGenerators

Input: r_1,r_2 partial reactions, G_1,G_2 partial reaction sets
Reference-passed: G: partial reaction set
  if $[r_2] \setminus [r_1] \neq \perp$ **then**
    for all $g \in G_1$ **do**
      if $g \bowtie r_1$ and $g \bowtie r_2$ and $[g] \wedge ([r_2] \setminus [r_1]) \neq \perp$ **then**
        if $[r_1 \vee g] = [r_2]$ **then** $G \leftarrow G \cup \{r_1 \vee r_2 \vee g\}$
        else MergeGenerators(r_2,$r_1 \vee g$,G_2,G_1,G)
  else MergeGenerators(r_1,r_2,G_1,G_2,G)

4.2.3 Weak endochrony check

A process is weakly endochronous as soon as we do not need the forced absence information to distinguish between elements of $\mathcal{G}(P,p)$.

Theorem 4 (Weak endochrony) *Let P be a process and let p be the set of its equations. Then, P is WE* iff *for all $p,q \in \mathcal{G}(P,p)$, $p \neq q$:*

$$\left.\begin{array}{c} supp(p) \cap supp(q) \neq \emptyset \\ [p],[q] \neq \bot \end{array}\right\} \Rightarrow \exists S : \mathcal{D}_S \ni p(S) \neq q(S) \in \mathcal{D}_S \qquad (WE - atoms)$$

In this case, the atom set of P is:

$$Atoms(P) = \{[g] \mid g \in \mathcal{G}(P,p), [g] \neq \bot\}$$

Proof sketch: If P is WE, then its atoms determine the generators. Each atom is a reaction, and therefore generated. Also if an atom is decomposed into more than one 1 generators, I have a contradiction with the minimality of atoms. Therefore, for each atom I can build the corresponding generator (which assigns other signals to $\perp\!\!\!\perp$). These generators have the needed properties.

If $\mathcal{G}(P,p)$ has property $(WE - atoms)$, then $\mathcal{R}(P)$ is generated by $\{[g] \mid g \in \mathcal{G}(P,p)\}$, and the properties of generators imply weak endochrony. $\diamond$

Checking the weak endochrony of a generator set is therefore easy. Moreover, when a system is not weakly endochronous, intuitive diagnostic messages are readily available. If $a,b \in \mathcal{G}(P,p)$, $a \neq b$, with $supp(a) \cap supp(b) \neq \emptyset$ and $supp([a]) \cap supp([b]) = \emptyset$, then $[a]$ and $[b]$ are reactions of P that are not independent, but cannot be distinguished in an asynchronous environment. The user must change the signalling protocol to make them contradictory, for instance by addition of supplementary communication lines or messages. Automatic synthesis of weak endochrony is possible, but doing it well is a difficult problem.

4.2.4 Examples

The generator set of the process in Fig. 4 is:

$$\{(I1^\top, O1^\top, C^{\perp\!\!\!\perp}, SYNC^{\perp\!\!\!\perp}), (I2^\top, O2^\top, C^{\perp\!\!\!\perp}, SYNC^{\perp\!\!\!\perp}),$$
$$(I1^\top, O1^\top, I2^\top, O2^\top, C^\top, SYNC^\top)\}$$

As expected, the process is not weakly endochronous because $[(I1^\top, O1^\top, I2^\top, O2^\top, SYNC^\top)]$ and $[(I1^\top, O1^\top, SYNC^{\perp\!\!\!\perp})]$ are neither conflicting, nor of disjoint support.

The generator set of the transformed process in Fig. 5 is:

$$\{(I1^\top, O1^\top, C^{\perp\!\!\!\perp}, SYNC1^0), (I2^\top, O2^\top, C^{\perp\!\!\!\perp}, SYNC2^0),$$
$$(I1^\top, O1^\top, I2^\top, O2^\top, C^\top, SYNC1^1, SYNC2^1)\}$$

The process is weakly endochronous.

5 Executive generation

Once we determined that a process is weakly endochronous we can generate for it the shell (executive) described in Section 2. We shall present here a simple technique for generating the shell directly from the atom set.

To further simplify code generation, we also assume here that the weakly endochronous process P describing our shell is I/O deterministic.[4] Formally, we shall assume that for any two different $a, b \in Atoms(P)$ we have:

$$a \not\bowtie b \Rightarrow a/\mathcal{I}(P) \not\bowtie b/\mathcal{I}(P)$$

where $r/\mathcal{I}(P)$ is the restriction of r on the input signals.

The executive we generate for P takes advantage of atom independence properties. For each atom a, independently, we generate a function that cyclically:

1. Awaits the arrival of a new value $a(S)$ for each signal $S \in supp(a/\mathcal{I}(P))$.
2. When all needed input arrived, trigger a reaction of the pearl, with $a/\mathcal{I}(P)$ as input, and then output the values of $a/\mathcal{O}(P)$.

The actual executive is formed of all the functions associated with the atoms, running in parallel. In practice, parallelism can be simulated using a simple event-driven approach, where each incoming event $S^v, v \in \mathcal{D}_S$ triggers computations in the functions associated with the atoms a having $a(S) = v$.

The actions of step 2 above are complex, and must be executed atomically. In a purely asynchronous shell, no two atoms must be executing step 2 at the same time (it is a mutual exclusion region). The operations performed during step 2 are the following:

1. Enter the mutual exclusion region.
2. Feed the data in $a/\mathcal{I}(P)$ to the pearl.
3. Trigger the computation of the pearl by enabling the clock (in hardware) or calling the reaction function (in software).
4. When the reaction is completed: In hardware, disable the clock. In software, update the pearl state.
5. Send the data in $a/\mathcal{O}(P)$. This may block the computation until output communication lines are free.
6. Signal the fact that the data in $a/\mathcal{I}(P)$ has been read. Remove these messages from the await list of other atoms. New signals can arrive on the input ports of $supp(a/\mathcal{I}(P))$.
7. Exit the mutual exclusion region.

[4] In general, weakly endochronous systems can be I/O non-deterministic, allowing for instance the specification of tests over infinite input data, abstracted as `event`.

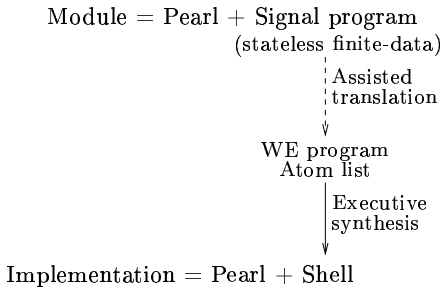

Figure 6: Assisted implementation flow

The previous operations can be easily be implemented in either software or hardware (using mutexes, semaphores, etc.). In both cases, mutual exclusion must be ensured to avoid races between the various functions. In hardware, such races may lead to metastability, or simply to erroneous functioning. In software, they can lead to deadlocks.

With the definition of the executives, we have completed the definition of the shell synthesis flow, which has the structure presented in Fig. 6.

Many implementations of the previously-defined executives are possible, in both hardware and software. Many optimizations and extensions are possible, too, which will be the subject of future work. We mention here only three:

- Hierarchize the waiting process, exploiting exclusiveness between atoms using successive tests over the input values. This corresponds to constructing a DAG structure having a similar function to the *clock tree* of Signal [1].
- Take into account some state information in the executives (some simple over-abstraction of the pearl state space).
- Mutual exclusion can be relaxed in implementations where the AFSM is actually implemented using synchronous technology, like in generalized latency-insensitive systems [17]. One possibility is to allow successive atoms to be pipelined in certain cases.

6 Composition issues

Consider the GALS system formed using the executives defined in the previous section. To ensure its correctness w.r.t. the specification, we still need to ensure some global synchronization properties, presented in this section.

6.1 Causality and completeness

In our approach, the causality of computations is determined by the pearl reactions, and by the ability of the shells to control the pearls. In a reaction of the pearl, all input must come before the computation takes place (clock enable/disable cycle in hardware, reaction function call in software), and before any output can be produced. The finest reactions, which give the causal dependencies, are the atoms.

By consequence, some reactions of the synchronous model may be unfeasible in the GALS implementation. Consider, for instance, the GALS system synthesized from the following specification:

```
process EXAMPLE2 =
(? event I ; ! event O ;)
(| (A,B) := P1(I) | O:=P2(A,B) |)
where event A,B ;
  process P1 =
  (? event I,A ; ! event B ;) (| A ^= B ^= I |);
  process P2 =
  (? event A ; ! event B,O ;) (| A ^= B ^= O |) ;
end ;
```

Each of the two sub-processes representing our modules has exactly one atom: $a_1 = (A^\top, B^\top, I^\top)$ for `P1`, and $a_2 = (A^\top, B^\top, O^\top)$ for `P2`. A synchronous reaction of `EXAMPLE2` is $a_1 \vee a_2 = (A^\top, B^\top, I^\top, O^\top)$, but the GALS system cannot perform it, because the dependency on signal `A` requires that a_1 is executed before a_2, whereas the dependency on `B` requires the inverse dependency.

We need to ensure that the GALS implementation indeed implements all the functions of the synchronous specification. To do this, we need to ensure that the atom system is acyclic in all possible input configurations. This problem is similar to the separate compilation problem of other synchronous languages such as Lustre [9], to the correctness check of cyclic definitions [13] (and the sub-case of constructive causality checks in Esterel).

In our case, a simple criterion ensuring the correctness of the GALS implementation is the acyclicity of the global atom system. More complex analyses are possible, but they will be the subject of future work.

6.2 Weak isochrony

The weak endochrony of the components ensures deterministic operation of the system. The acyclicity ensures that all specified behaviors of the system are indeed feasible. We want to ensure now that no other behaviors are feasible than those specified. More precisely, that any execution of the GALS implementation is a sub-set of a synchronous behavior of the specification.

This is not always the case. Consider, for instance, the GALS system synthesized from the following specification:

```
process EXAMPLE2 =
(? boolean I ; ! event O ;)
(| (A,B) := PROD(I) | O:=CONS(A,B) |)
where event A,B ;
  process PROD =
  (? boolean I ; ! event A,B ;)
  (| A:=when I=0 | B:=when I=1 |);
  process CONS =
  (? event A,B ;) (| A^=B |) ;
end ;
```

The atoms of PROD are $(I^0, A^\top)$ and $(I^1, B^\top)$. The only atom of CONS is $(A^\top, B^\top)$. The only behavior of process EXAMPLE2 is $\perp$. Therefore, it has no atom.

However, if PROD and CONS are compiled using the technique of Section 5, the resulting GALS system has non-void asynchronous behaviors. For instance, after reading I^0 and I^1, it produces $O^\top$. This is due to the fact that the events $A^\top$ and $B^\top$ are produced in different reactions of PROD, but read by CONS in a single reaction. The reverse can also happen, where signals produced synchronously are read in different instants.

The criterion ensuring that such stray behaviors cannot occur is *weak isochrony*. After a few notations, we give the basic definition of Potop, Caillaud, and Benveniste [16], as simplified by our stateless framework.

Definition 4 (asynchronous prefixes) *Let P be a process. Then, $Head(P)$ is the set of all $h \in \mathcal{R}$ such that there exists $t \in \mathcal{T}(P)$ with $h(S)$ being the first value on signal S in t, or $\perp$, if no such value exists.*

In other terms, $Head(P_i)$ is the set of all asynchronous prefixes of depth 1 of executions of P. With this notation:

Definition 5 (weak isochrony) *Let P be a process with sub-processes $P_1, \ldots, P_n$. We say that the processes $P_1, \ldots, P_n$ are weakly isochronous if, by definition, any execution of the GALS implementation can be started with a synchronous reaction.*

Formally, for all $r_i \in Head(P_i)$, such that any two r_i, r_j are equal on $\mathcal{V}(P_i) \cap \mathcal{V}(P_j)$ we have:

$$\exists \overline{r}_i \in \mathcal{R}(P_i), 1 \leq i \leq n : \begin{cases} \overline{r}_i \leq r_i \\ \bigvee_{i=1}^{n} \overline{r}_i \in \mathcal{R}(P) \\ \bigvee_{i=1}^{n} \overline{r}_i \neq \perp \end{cases}$$

Checking weak isochrony in its basic form is decidable, but can be very complex, due to the computation of $Head(P_i)$. We propose here a sufficient property adapted to our atom-based approach.

Theorem 5 (sufficient conditions) *Let P be a process with sub-processes $P_1, \ldots, P_n$. Then, a sufficient condition for $P_1, \ldots, P_n$ to be isochronous is that the input of no atom of a component can be covered by atoms of other components, unless these atoms are independent.*

Formally: We denote with $\mathcal{A}_i$ the union of all $Atoms(P_j)$ with $j \neq i$. Then, $P_1, \ldots, P_n$ are isochronous if: For all $1 \leq i \leq n$, and all $a \in Atoms(P_i)$, if $a_1, \ldots, a_k \in \mathcal{A}_i$ such that for all $S \in supp(a/\mathcal{I}(P))$ there exists j with $a_j(S) = a(S)$, then $supp(a_i)$ are mutually disjoint.

Proof sketch: Consider $r_i \in Head(P_i), 1 \leq i \leq n$ satisfying the hypothesis of Definition 5. There exists i and $a \in Atoms(P_i)$ such that $a \leq r_i$. Using the hypothesis of the theorem, we can incrementally build a common reaction by appending atoms to one or the other of the P_j. $\diamond$

Note that the criterion offered by Theorem 5 offers a simple computational criterion based on the atoms already computed in previous sections. Determining isochrony is done through a simple traversal of atom covers. The criterion is an over-approximation of weak isochrony. It can be improved, for instance by taking into account causality information.

7 Conclusion

We have defined a method for synthesizing the asynchronous executives that are driving the synchronous modules of a (GALS) system. The technique takes as input high-level synchronization constraints. The resulting GALS system is predictable and functionally correct and complete with respect to the initial synchronous specification, and regardless of the size of the communication lines. The technique allows the synthesis of executives for all specifications whose modules are stateless weakly endochronous.

Future work. The current paper only concerns correctess in an untimed setting. Our long-term objective is to take into account and/or guarantee real-time requirements, such as periodicity, end-to-end, or throughput constraints. This involves the definition of timing analysis and scheduling techniques compatible with our executives. On the other hand, the executives could be simplified under specific timing hypothesis (for instance, the FIFO protocols can be simplified if the reader is faster than the writer).

The various steps of our technique can be improved and extended in a variety of ways, already mentioned in Sections 5 and 6.

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] 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. Butazzo. *Hard real-time computing systems. Predictable scheduling, algorithms and applications.* Kluwer, 2002.

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

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

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

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

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

[13] K. Namjoshi and R. Kurshan. Efficient analysis of cyclic definitions. In *Proceedings CAV'99*, pages 394–405, 1999. LNCS1633.

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

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

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

[17] 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