Fundamenta Informaticae 107 (2011) 1–28
DOI 10.3233/FI-2011-300
IOS Press

From Concurrent Multi-clock Programs to Deterministic Asynchronous Implementations

Dumitru Potop-Butucaru* **and Yves Sorel**
INRIA Rocquencourt, Domaine de Voluceau, BP105, 78153 Le Chesnay Cedex, France
dumitru.potop@inria.fr, yves.sorel@inria.fr

Robert de Simone
INRIA Sophia Antipolis, 2004, route des Lucioles, 06902 Sophia Antipolis Cedex, France
robert.de_simone@inria.fr

Jean-Pierre Talpin
INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
jean-pierre.talpin@inria.fr

Abstract. We propose a general method to characterize and synthesize correctness-preserving asynchronous wrappers for synchronous processes on a globally asynchronous locally synchronous (GALS) architecture. While a synchronous process may rely on the absence of a signal to trigger a reaction, sensing absence in an asynchronous environment may be unfeasible due to uncontrolled communication latencies. A simple and common solution is to systematically encode and send absence notifications, but it is unduly expensive at run-time. Instead, our approach is based on the theory of weakly endochronous systems, which defines the largest sub-class of synchronous systems where (possibly concurrent) asynchronous evaluation is faithful to the original (synchronous) specification. Our method considers synchronous processes or modules that are specified by synchronization constraints expressed in a high-level multi-clock synchronous reactive formalism. The algorithm uses a compact representation of the abstract synchronization configurations of the analyzed process and determines a minimal set of synchronization patterns generating by union all its possible reactions. A specification is weakly endochronous if and only if these generators do not need explicit absence information. In this case, the set of generators can directly be used to synthesize the concurrent asynchronous multi-rate wrapper of the process.

Keywords: endochrony, synchrony, multi-clock, asynchronous implementation

*Address for correspondence: INRIA Rocquencourt, Domaine de Voluceau, BP105, 78153 Le Chesnay Cedex, France

1. Introduction

Synchronous programming is nowadays a widely accepted paradigm for the design of critical applications such as digital circuits or embedded software [3], especially when a semantic reference is sought to ensure the coherence between the implementation and the various simulations. The synchronous paradigm supports a notion of *deterministic concurrency* which facilitates the functional modeling and analysis of embedded systems.

While modeling a synchronous process or module can be easy, implementing a concurrent system by composing synchronous modular specifications is often hardened by the need of preserving global synchronizations in the model of the system. These synchronization artifacts need most of the time to be preserved, at least in part, in order to ensure functional correctness when the behavior of the whole system depends on properties such as the arrival order of events on different channels, or the presence or absence of an event at a certain instant.

We address this issue and focus on the characterization and synthesis of wrappers that control the execution of synchronous processes in a GALS architecture. Our aim is to preserve the functional properties of individual synchronous processes deployed on an asynchronous execution environment. To this aim, we shall start by considering a multi-clocked or polychronous model of computation and lay the proper theoretical background to finally establish properties pertaining on the assurance of asynchronous implementability.

Our technique is mathematically founded on the theory of *weakly endochronous systems*, due to Potop, Caillaud, and Benveniste [18]. Weak endochrony gives a compositional sufficient condition establishing that a concurrent synchronous specification exhibits no behavior where information on the absence of an event is needed. Thus, the synchronous specification can safely be executed with identical results in any asynchronous environment (where absence cannot be sensed). Weak endochrony thus gives a latency-insensitivity and scheduling-independence criterion.

In this paper, **we propose the first general method to check weak endochrony on multi-clock synchronous programs**. The method is based on the construction of so-called *generator sets*. Generator sets contain minimal synchronization patterns that characterize all possible reactions of a multi-clocked program. These sets are used to check that a specification is indeed weakly endochronous, in which case they can be used to generate the GALS wrapper. In case the specification is not weakly endochronous, the generators can be used to generate intuitive error messages. Thus, we provide an alternative to classical compilation schemes for multi-clock programs, such as the clock hierarchization techniques used in Signal/Polychrony [1].

Outline. The paper is organized as follows: Sections 2 and 3 give an intuition of the problem addressed in this paper, together with references to previous work and an idea of the desired solution. Section 4 defines the formalism that will support our presentation. Section 5 summarizes the original theory of [18], adapts it to our framework, and provides novel algorithms allowing the manipulation of generator sets for weakly endochronous specifications. Section 6 provides the extension to general synchronous specifications, and defines novel algorithms to determine if the specification is weakly endochronous. We conclude in Section 7.

2. Multi-clock synchronous system

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-synchronized mode is done using the `SYNC` signal. The carry between the two adders is propagated through the Boolean wire `C` whenever `SYNC` is present. To simplify figures and notations, we group both integer inputs of `ADD1` under `I1`, and both integer inputs of `ADD2` under `I2`. This poses no problem because from the synchronization perspective of this paper the two integer inputs of an adder have the same properties.

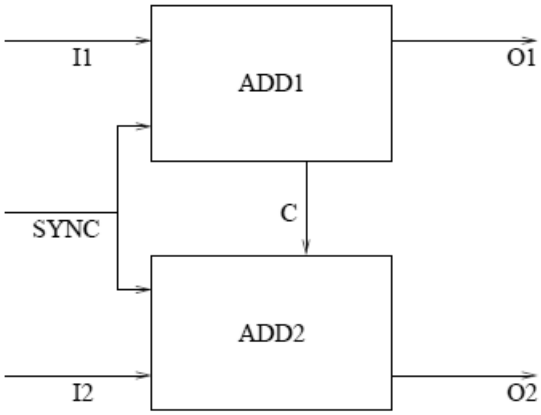

Figure 1. Data-flow of a configurable adder.

We consider a discrete model of time, where executions are sequences of *reactions*, indexed by a *global clock*. Given a synchronous specification (also called *process*), a reaction is a valuation of the *input, output and internal (local) signals* of the process. Fig. 2 gives a possible execution of our example. We shall denote with $\mathcal{V}(\ )$ the finite set of signals of a process . We shall distinguish inside $\mathcal{V}(\ )$ the disjoint sub-sets of *input* and *output signals*, respectively denoted $\mathcal{I}(\ )$ and $\mathcal{O}(\ )$.

If we denote with `EXAMPLE` our configurable adder, then

$$\begin{aligned}\mathcal{V}(\texttt{EXAMPLE}) &= \{\texttt{I1 I2 SYNC O1 O2 C}\}\\ \mathcal{I}(\texttt{EXAMPLE}) &= \{\texttt{I1 I2 SYNC}\}\\ \mathcal{O}(\texttt{EXAMPLE}) &= \{\texttt{O1 O2}\}\end{aligned}$$

All signals are typed. We denote with $\mathcal{D}_S$ the domain of a signal . 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 $\mathcal{D}_S$. Otherwise, we say that it is *absent*. Absence is simply represented with a special value $\perp$, which is appended to all domains $\mathcal{D}_{\mathtt{S}}^{\perp} = \mathcal{D}_{\mathtt{S}} \cup \{\perp\}$.

Formally, a reaction of a process is a valuation of all the signals of $\mathcal{V}(\ )$ into their extended domains $\mathcal{D}_{\mathtt{S}}^{\perp}$. We denote with $\mathcal{R}(\ )$ the set of all reactions of . Given a set of signals $\mathcal{V}$, we denote with $\mathcal{R}(\mathcal{V})$ the set of all possible valuations of the signals in $\mathcal{V}$. Obviously, $\mathcal{R}(\ ) \subseteq \mathcal{R}(\mathcal{V}(\ ))$.

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	$\perp$	$\perp$	$\bullet$	$\perp$	$\perp$	$\bullet$	$\perp$
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

Figure 2. A synchronous run of the adder

The *support* of a reaction , denoted ( ), is the set of present signals. For instance, the support of reaction 4 in Fig. 2 is { 1 2 1 2}. In a reaction of process , we distinguish the *input event*, which is the restriction $|_{\mathcal{I}(P)}$ of to input signals, and the *output event*, which is the restriction $|_{\mathcal{O}(P)}$ to output signals.

In many cases we are only interested in the presence or absence of a signal, because it transmits no data, just synchronization (or because we are only interested in synchronization aspects). To represent such signals, the Signal language [11] uses a dedicated type named `event` of domain $\mathcal{D}_{\texttt{event}} = \{\bullet\}$. We follow the same convention: In our example, `SYNC` has type `event`. In Fig. 2, the signal types are `SYNC:event`; `O1,O2:integer`; `I1,I2:integer_pair`; `C:boolean`.

To represent reactions, we use a *set-like convention* and omit signals with value $\perp$. Reaction 4 is denoted $(\texttt{I1}^{(9,9)}\ \texttt{O1}^{8}\ \texttt{I2}^{(0,0)}\ \texttt{O2}^{0})$.

3. Deterministic asynchronous implementation

We consider a synchronous process, and we want to execute it in an asynchronous environment where inputs arrive and outputs depart via asynchronous FIFO channels with uncontrolled (but finite) communication latencies. To simplify, we assume that we have exactly one channel for each input and output signal of the process. We also assume a very simple correspondence between messages on channels and signal values: Each message on a channel corresponds to exactly one value (not absence) of a signal in a reaction. In particular, no message represents absence.

The execution machine driving the synchronous process in the asynchronous environment cyclically performs the following 3 steps:

1. assembling asynchronous input messages arriving onto the input channels into a synchronous input event acceptable by the process,
2. triggering a reaction of the process for the reconstructed input event, and
3. transforming the output event of the reaction into messages onto the output asynchronous channels.

Fig. 3 provides the general form of such an execution machine, which is basically a wrapper transforming the synchronous process into a globally asynchronous, locally synchronous (GALS) [7] component that

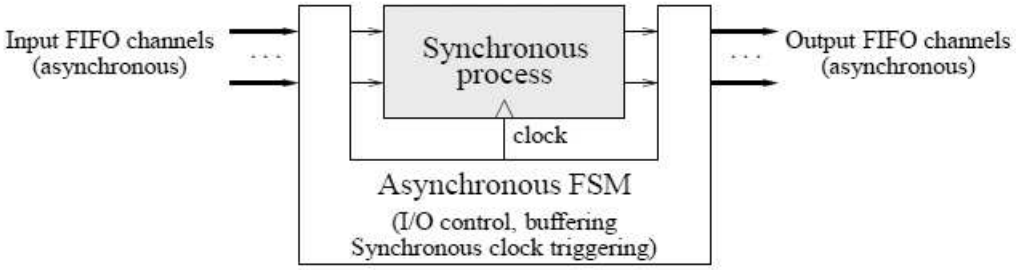

Figure 3. GALS wrapper driving the execution of a synchronous process in an asynchronous environment

can be used in a larger GALS system. The actual form of the asynchronous finite state machine (AFSM) implementing the execution machine, and the form of the code implementing the synchronous process largely depends on a variety of factors, such as the desired implementation (software or hardware), the asynchronous signaling used by the input and output FIFOs, the properties of the synchronous process, etc.

In order to achieve deterministic execution,[1] the main difficulty lies in step (1) above, as it involves the potential reconstruction of signal absence, whereas absence is meaningless in the chosen asynchronous framework. 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 Fig. 4, that we consider the inputs of Fig. 2 without synchronization information.

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

Figure 4. Corresponding asynchronous run. No synchronization exists between the various signals, so that correctly reconstructing synchronous inputs from the asynchronous ones is impossible

The adder `ADD1` will then receive the first value $(1\ 2)$ on the input channel `I1` and $\bullet$ on `SYNC`. Depending on the arrival order, which cannot be determined, any of the reactions $(\texttt{I1}^{(1,2)}\ \texttt{O1}^{3}\ \texttt{SYNC}^{\bullet}\ \texttt{C}^{0})$ or $(\texttt{I1}^{(1,2)}\ \texttt{O1}^{3})$ can be executed by `ADD1`, leading to divergent computations. The problem is that these two reactions are not independent, but no value of a given channel allows to differentiate one from the other (so one can't deterministically choose between them in an asynchronous environment).

Deterministic input event reconstruction is therefore impossible for some synchronous processes. This means that a methodology to implement synchronous processes on an asynchronous architecture

[1] Like in [17], determinism can be relaxed here to predictability – the fact that the environment is always informed of the choices made inside the process. While this involves only small changes in the following technical results, we preferred a simpler presentation.

must rely on the (implicit or explicit) identification of some class of processes for which reconstruction is possible. Then, giving a deterministic asynchronous implementation to any given synchronous process is done in two steps:

Step 1. Transforming the initial process, through added synchronizations and/or signals, so that it belongs to the implementable class.

Step 2. Generating an implementation for the transformed process.

The choice of the class of implementable processes is therefore essential. On one hand, choosing a small class can highly simplify analysis and code generation in step (2). On the other, small classes of processes result in heavier synchronization added to the process in step (1). Our choice, justified in the next section, is the class of weakly endochronous processes.

3.1. Previous Work and Motivation

The most developed notions identifying classes of implementable processes are the concepts of *latency-insensitive systems* of Carloni *et al.* [5] and the *endochronous systems* of Benveniste *et al.* [2, 11].

The latency-insensitive systems are those featuring no signal absence. Transforming processes featuring absence, such as our example of Figures 1 and 2, into latency-insensitive ones amounts to transforming the presence/absence of a signal into a true/false value that is sent and received as an asynchronous message. This is easy to check and implement, but often results in an unneeded communication overhead due to the absence messages. Several variations and hardware implementations of the theory have been proposed, of which we mention here only the one by Vijayaraghavan and Arvind [22].

The endochronous systems and the related hardware-centric *generalized latency-insensitive systems* [21] are those where the presence and absence of all signals can be incrementally inferred starting from the state and from signals that are always present. For instance, Fig. 5 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 communicating (`C` absent), and 3 for the synchronized execution of the two adders (`C` present). Note that the value of `SYNC` determines the presence/absence of all signals.

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

Figure 5. Endochronous solution

Checking endochrony consists in ordering the signals of the process in a tree representing the incremental process used to infer signal presence (the signals that are always read are all placed in the tree root). The compilation of the Signal/Polychrony language is currently founded on a version of endochrony [1].

The endochronous 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. This means that no concurrency remains between sub-processes of an endochronous process. For instance, in the endochronous model of our adder, the behavior of the two adders is synchronized at all instants by the `SYNC` signal (whereas in the initial model the adders can function independently whenever `SYNC` is absent). By consequence, using endochrony as the basis for the development of systems with internal concurrency has 2 drawbacks:

- Endochrony is non-compositional (synchronization code must be added even when composing processes sharing no signal).
- Specifications and implementations/simulations are often over-synchronized.

Weak endochrony, due to Potop, Caillaud, and Benveniste [18] and presented in Section 5, generalizes endochrony by allowing both synchronized and non-synchronized (independent) computations to be realized by a given process. 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 [15]), so that the first one does not discard the second. This is also linked to the Kahn principles for networks [12], where only internal choice is allowed to ensure that overall lack of confluence cannot be caused by input signal speed variations.

Fig. 6 presents a run of a weakly endochronous system obtained by replacing the `SYNC` signal of our example with two input signals:

- `SYNC1`, of Boolean type, is received at each execution of `ADD1`. It has value 0 to notify that no synchronization is necessary, and value 1 to notify that synchronization is necessary and the carry signal `C` must be produced.
- `SYNC2`, of Boolean type, is received at each execution of `ADD2`. It has value 0 to notify that no synchronization is necessary, and value 1 to notify that synchronization is necessary and the carry signal `C` must be read.

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

Figure 6. Weakly endochronous solution.

The two adders are synchronized when SYNC1=1 and SYNC2=1, corresponding to the cases where SYNC=• in the original design. However, the adders function independently elsewhere (between synchronization points).

From a practical point of view, weak endochrony supports less synchronized, concurrent GALS implementations. While the implementation of latency-insensitive and endochronous synchronous processes is strictly bound by the scheme of Fig. 3, wrappers of weakly endochronous processes may exploit the concurrency of the specification by directly and concurrently activating various parts of a process. In the context of the example of Fig. 6, the GALS wrapper may consist of an AFSM that can independently activate the two adders, the activations being synchronized only when SYNC1=SYNC2=1.

3.2. Related work

In addition to previously-mentioned existing work, which perfectly fit in our classification, many other approaches exist to the generation of deterministic asynchronous implementations of synchronous specifications. We mention only two of them.

The *desynchronization* approach of Cortadella *et al.* [8] takes as input a synchronous circuit and produces an asynchronous circuit simulating its execution. From the point of view of our classification of Section 3.1, desynchronization is closely related to latency insensitivity: *Signal absence* has no semantic existence, so that the asynchronous simulation will compute (and communicate) one value for each register and each simulated synchronous instant. Like in latency-insensitive design, focus here is on the asynchronous handshaking protocols, and the authors define conditions ensuring that no dead-locks occur.

As Cortadella *et al.* note, techniques based on the endochrony property (and therefore the results of this paper) could be used in a desynchronization framework to improve performance and lower power consumption. However, our paper does not advance in this direction.

The OCRep tool by Caspi, Girault *et al.* [6] takes as input a synchronous program and produces a asynchronous semantics-preserving distributed implementation. As in endochrony-related approaches, the objective is the minimization of communications. The difference is that the approach is holistic, whereas endochrony-based approaches aim at incremental development processes where composition is central.

4. Multi-clock Specification in Signal

The use of weakly endochronous processes allows the preservation of the independence of non-synchronized computations,[2] while adding the supplementary synchronization needed to ensure deterministic execution in an asynchronous environment. Weak endochrony is preserved by synchronous composition, thus supporting incremental development. However, the lack of a practical technique for checking and/or synthesizing weak endochrony limited its use in practice until now.

We use the high-level multi-clock synchronous data-flow language Signal [1] to demonstrate the applicability of our technique. This language allows a simple representation of the clock synchronization constraints we are interested in. Like other synchronous data-flow formalisms, such as Lustre, Scade,

[2] So that later analysis or implementation steps can exploit it.

Lucid, that could also have been considered, Signal gives an implicit representation of states that is most convenient (yet not mandatory) for a direct illustration of our technique.

4.1. Finite stateless abstraction

We define our decision procedure for weak endochrony on the finite-data stateless abstraction of Signal programs that is already used in existing compilers. This subset is defined by (1) a restriction to finite data types and (2) the abstraction of delay equations (sole to introduce implicit state transition) by synchronization constraints (between the signals of a delay equation).

For programs featuring infinite data and delays (e.g., `integer`, `float`) the construction of an finite-data stateless abstraction is done by a procedure of the Signal compiler that is detailed in [16]. Given that a Signal specification needs not be functionally complete, the abstraction can be represented as a Signal process (and it is derived through simple transformations of the Signal source).

The stateless abstraction does not mean all state information is lost. The abstraction procedure automatically conserves some of the underlying synchronization information, and the programmer can force the preservation of as much synchronization information as needed through the addition of so-called *clock constraints* (defined in Section 4.3.1), which are preserved by the abstraction procedure. For instance, activation conditions such as the ones used in the compilation of Esterel [4] can be easily preserved in this way.

However, the abstraction means that: (1) Certain weakly endochronous processes are rejected, as the analysis cannot determine that the property holds[3] and (2) The code generated for a weakly endochronous process may be over-synchronized. We shall give more details on abstraction-related issues later in the paper.

4.2. Process structure

In Signal, a specification is a *process*, whose definition may involve other processes, hierarchically, through a mechanism called *sub-process instantiation* which is similar to the use of templates in classical imperative languages such as C. Fig. 7 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 data-flow specification, and a local declaration section. In our example, the top-level 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 "!". Our example has no state, and the infinite type signals (`I1`, `I2`, `O1`, `O2`) have been replaced with signals of type `event` by the abstraction procedure.

The Boolean type of the carry `C` has also been transformed into `event`, because it is computed from `I1` (we need to preserve determinism). The carry signal `C` is internal to our process, so its declaration is not part of the process interface.

The data-flow specification of `EXAMPLE` consists of two equations, which define the interconnections between `ADD1`, `ADD2`, and the environment. We say that `EXAMPLE` instantiates `ADD1` and `ADD2`. The local definition section defines the internal signal `C`, and the processes `ADD1` and `ADD2`. The hierarchy of processes allows the structuring of the specification and the definition of signal scopes that mask internal signals. Process `EXAMPLE` using process `ADD1` in its data-flow intuitively corresponds to replacing the

[3] For instance, because the integer signal used to choose between two reactions has been abstracted away and replaced with a signal having only a present or absent status.

```
1  process EXAMPLE = (? event SYNC,I1, I2
2                     ! event O1, O2 )
3  (| (O1,C) := ADD1 (SYNC,I1)
4   | O2 := ADD2 (SYNC,I2,C)
5   |)
6  where
7    event C ;
8    process ADD1 = (? event SYNC, I1
9                    ! event O1, C )
10   (| I1   ^= O1
11    | SYNC ^< I1
12    | C    ^= SYNC |) ;
13   process ADD2 = (? event SYNC, I2, C
14                   ! event O2 )
15   (| I2   ^= O2
16    | SYNC ^< I2
17    | C    ^= SYNC |) ;
18 end ;
```

Figure 7. The Signal process of the configurable adder in Fig. 1

instance of ADD1 in EXAMPLE with the data-flow of ADD1. Subprocess instantiation is fully defined in Section 4.4.

4.3. Data-flow

The data-flow specification of a process is formed of *equations* defining *constraints* between the signals of the process. There are two types of equations: clock constraints and assignments. The use of a constraint language allows us to easily manipulate functionally incomplete specifications.

Recall from Section 2 that the set of signals of a process is denoted $\mathcal{V}(\ )$, and that the sub-sets of input and output signals are respectively denoted $\mathcal{I}(\ )$ and $\mathcal{O}(\ )$. The set $\mathcal{R}(\ )$ of reactions of a stateless Signal process is, by definition, the set of all reactions over $\mathcal{V}(\ )$ that satisfy all the the constraints represented by statements of , be them clock constraints or assignments.

4.3.1. Clocks and 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*. Lines 10 to 12 of Fig. 7, which give the constraints of ADD1, illustrate 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 the synchronization signal is only present when ADD1 performs some computation on inputs. Formally, we require that the presence of SYNC

implies the presence of I1, or that the clock ^SYNC is included in (smaller than) the clock ^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. Clock exclusion is not used in our example. Writing "S1^#S2" requires that signals S1 and S2 are never present in the same reaction.

4.3.2. 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, "X:=Y$ init V" [4], is simply abstracted by its synchronization requirement "X^=Y". 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 clock and value identity of X and Y. Signal Y can also be replaced with a data-flow expression built using the following operators:

The operator when performs conditional down-sampling. The signal "X when C" is equal to X whenever X is present and 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. 8, "when SYNC1=1" is a signal of type event that is present when signal SYNC1 is present with value 1. 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.

4.4. Subprocess instantiation syntax and semantics

The subprocess instantiation mechanism of Signal uses a simple syntax to allow a process to include all the statements of another process, modulo some variable renaming. We say that the first process *instantiates* the second one.

Given a process with input signals $_1$ $_n$ and output signals $_1$ $_m$, it can be instantiated by another process by using in a modified assignment equation:

$$(\ _1 \quad _m) := \ (\ _{m+1} \quad _{m+n})$$

where

- The function name is replaced with the name of the instantiated process (, in our example).
- The function arguments are replaced by the signals of corresponding to the inputs of . In our example, signal $_{m+i}$ corresponds to $_i$ for $1 \leq \ \leq n$.
- The assigned value is replaced with a tuple of signals of corresponding to the outputs of . In our example, signal $_i$ corresponds to $_i$ for $1 \leq \ \leq$.

The signals $_1$ $_{m+n}$ need not be distinct.

The semantics of is as if the instantiation statement given above is replaced in the body of by all the statements of , modified as follows:

[4] The value of X is V the first time Y occurs, and then takes the previous value of Y.

- Each occurrence of $_i$ is replaced by $_{m+i}$ for $1 \leq \ \leq n$.
- Each occurrence of $_i$ is replaced by $_i$ for $1 \leq \ \leq \ $.
- For each internal signal of , a new internal signal $'$ is defined in , of the same type as , but with a name different from all other signals of . Each occurrence of is replaced by $'$.

For instance, in Fig. 8, the instantiation of `EXAMPLE` by `EXAMPLE2` will replace all occurrences of signal `SYNC` with `SYNC_AUX` according to the instantiation statement of line 6. The other signals of the example keep their name, which does not changes from `EXAMPLE` to `EXAMPLE2`.

4.4.1. Notations

The statements replacing inside are called the expansion of in . We denote with $[X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k]$ or $[X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ]$ the expansion of process which renames signal X_i to $_i$ for $1 \leq \ \leq \ $. The set of signals of this expansion is, by definition:

$$\mathcal{V}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k]) = (\mathcal{V}(\ ) \setminus \{X_i \mid 1 \leq \ \leq n\}) \cup \{\ _i \mid 1 \leq \ \leq n\}$$

Consider a reaction $\in \mathcal{R}(\ )$. We naturally want to assign it a corresponding reaction over the transformed set of signals. But this is only possible when is compatible with the signal renaming. More precisely, whenever two different signals X_i and X_j ($\neq$) of are assigned through expansion the same signal name $_i = \ _j$, transforming is only possible when $(X_i) = \ (X_j)$. We shall denote the transformed reaction $[X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k]$. We denote with $\mathcal{R}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$ the set of all transformed reactions. These are all the reactions over $\mathcal{V}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$ that are compatible with all the constraints of $[X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k]$.

Consider now a set of signals $\mathcal{V}$ such as $\mathcal{V} \supseteq \mathcal{V}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$. We define the set $\mathcal{R}^{\mathcal{V}}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$ of reactions over $\mathcal{V}$ that equal a reaction of $\mathcal{R}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$ on $\mathcal{V}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$ and have any value (present or absent) on the remaining signals. These are all the reactions over $\mathcal{V}$ that are compatible with all the constraints of $[X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k]$.

When $\in \mathcal{R}(\mathcal{V})$ and $\mathcal{V}' \supseteq \mathcal{V}$, we define $|_{\mathcal{V}'}$ (an extension of the restriction notation of Section 2) as the reaction of $\mathcal{R}(\mathcal{V}')$ that equals over $\mathcal{V}$ and $\perp$ elsewhere. Then, in the context of previous definitions, if $\in \mathcal{R}(\ )$ and $\mathcal{V}' \supseteq \mathcal{V}(\ )$, we have:

$$[X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k] \ |_{\mathcal{V}'} \in \mathcal{R}^{\mathcal{V}'}(\ [X_1 \leftarrow \ _1 \quad X_k \leftarrow \ _k])$$

5. Weak endochrony

The theory of *weakly endochronous (WE) systems* [18], 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. 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 [17], the axioms of weak endochrony can be traced down to the fundamental result of

Keller [13] on the deterministic operation of a system in an asynchronous environment. Moreover, WE systems are synchronous Kahn processes, and weak endochrony extends to a synchronous framework the classical trace theory [14].

Absence not being needed in computations means that reactions sharing no common present value can be executed *independently* (without any synchronization). Absence is treated as a *don't care value* imposing no synchronization constraint (as opposed to present values).

```
1  process EXAMPLE2 = (? boolean SYNC1, SYNC2;
2                        event I1,I2
3                      ! event O1,O2 )
4  (| when SYNC1 ^= when SYNC2
5   | SYNC_AUX := when SYNC1
6   | (O1,O2) := EXAMPLE (SYNC_AUX, I1, I2)
7   |)
8  where
9    event SYNC_AUX ;
10   process EXAMPLE = the process in Fig. 7
11 end
```

Figure 8. A weakly endochronous refinement of process EXAMPLE

This property suggests a natural organization of the possible values of a signal as a Scott domain defined by $\perp \leq$, for all $\in \mathcal{D}_S$. The domain structure on particular signals induces a product partial order $\leq$ on reactions with $_1 \leq _2$ if and only if $(_1) \subseteq (_2)$ and $_1(\ ) = _2(\ )$ for all $\in (_1)$.

We say of two reactions $_1$ and $_2$ that they are *non-contradictory*, written $_1 \ _2$, if $_1(\ ) = _2(\ )$ for all $\in (_1) \cap (_2)$. Otherwise, we say that the reactions are *contradictory*, written $_1 \not\ _2$. Given a set of reactions , we shall say that it is non-contradictory, denoted if any two reactions of are non-contradictory.

The least upper bound and greatest lower bound induced by the order relation are respectively denoted with $\vee$ and $\wedge$, and called union and intersection of reactions. If $_1 \ _2$, both $_1 \vee _2$ and $_1 \wedge _2$ are defined, and we can also define the difference $_1 \setminus _2$, which has support $(_1) \setminus (_2)$ and equals $_1$ on its support. For a set with we denote $\vee = \bigvee_{r \in K}$.

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 is weakly endochronous if its set of reactions $\mathcal{R}(\ )$ is closed under the operations associated to the previously-defined domain structure: intersection, union, and difference of non-contradictory reactions.

(State-less) weak endochrony is compositional: the synchronous composition of two weakly endochronous systems is weakly endochronous. In fact, the results of [20] also suggest that the stateless weakly endochronous programs form the largest class of stateless synchronous programs with deterministic asynchronous implementations that is closed under synchronous composition.

When a synchronous program is not weakly endochronous, we can enforce weak endochrony by either restricting the behavior of the program, or by introducing new input signals whose values (and not present/absent status) can be used to make all the choices of the program. The second choice is taken in Fig. 8, where we take the example of Section 2 (Signal code in Fig. 7) and make it weakly endochronous by introducing two new Boolean inputs SYNC1 and SYNC2. A trace of the resulting program EXAMPLE2 is given in Fig. 6. Input SYNC1 is present in instants where ADD1 works. It has value 1 when ADD1 communicates its carry to ADD2, and value 0 otherwise. Similarly, SYNC2 is present in instants where ADD2 works. It has value 1 when ADD2 receives the carry from ADD1, and value 0 otherwise. This means that SYNC1 is true whenever SYNC2 is true, which is required by the clock constraint of Fig. 8. The signal SYNC of the original program is then computed in signal SYNC_AUX with "`when SYNC1`".

5.1. Atoms

From our point of view oriented towards automated analysis, it is most interesting that any behavior of a WE system can be decomposed into *atomic transitions*, or *atoms*. Formally, the set of atomic reactions of , denoted A ( ) is the set of the smallest (in the sense of $\leq$) reactions of $\mathcal{R}(\ )$ different from $\perp$. The set of atomic transitions is characterized by two fundamental properties: non-interference and generation.

Theorem 5.1. (atom set characterization)
A stateless process is weakly endochronous if and only if there exists a set of reactions $A \subseteq \mathcal{R}(\ )$ such that:

- *Generation:* The union of non-interfering atoms generates all the reactions of $\mathcal{R}(\ )$: $\mathcal{R}(\ ) = \{\bigvee \mid \subseteq A \wedge \}$.
- *Non interference:* Two distinct atoms $_1\ _2 \in A$, $_1 \neq\ _2$ either are contradictory or have disjoint support (in the latter case we shall say that they are independent).

(the proof of the theorem is given in reference [18])

Axiom (Non interference) implies that as soon as two atoms are not independent, they can be distinguished by a present value (not absence), meaning that choice between them can be done in an asynchronous environment.

The characterization of Theorem 5.1 corresponds to the case where no distinction is made between input, output and internal signals of a system (which is the case in [18]). As we seek to obtain deterministic asynchronous implementations for Signal programs, we require that the choice between any two contradictory atoms can be done based on input signal values.[5] Formally:

- *Input choice:* For any two contradictory atoms $_1\ _2 \in A$, there exists $\in \mathcal{I}(\ )$ such that $_i(\ ) \neq \perp\ \ = 1\ 2$, and $_1(\ ) \neq\ _2(\ )$.

The atom set of the weakly endochronous process in Fig. 8 is:

$$\{(\ 1^\bullet\ \ 1^\bullet\ \ 1^0)$$
$$(\ 2^\bullet\ \ 2^\bullet\ \ 2^0)$$
$$(\ 1^\bullet\ \ 1^\bullet\ \ 2^\bullet\ \ 2^\bullet\ \ 1^1\ \ 2^1\ \ ^\bullet\ \ \_A\ \ X^\bullet)\}$$

[5] To achieve predictability, choice can be done on input or output signal values.

Note that the first two atoms can be united to form the reaction where both `ADD1` and `ADD2` compute without exchanging a carry. The three atoms also satisfy the input choice property, so that a deterministic asynchronous implementation can be built.

Given a signal set $\mathcal{V}$, we shall denote with $A\quad(\mathcal{V})$ the set of atoms of any process with signal set $\mathcal{V}$ and no statement (*i.e.* no constraint). Such a degenerated process is weakly endochronous, with atom set:

$$A\quad(\mathcal{V}) = \{(X^v) \mid X \in \mathcal{V} \wedge \quad \in \mathcal{D}_X\}$$

However, does not satisfy the input choice property whenever it has outputs or internal variables of types different from `event`.

5.2. Atom-based implementation

We explained in Section 3 that our objective is the definition of a new implementation technique for the Signal language, based on the notion of weak endochrony. The key point of the approach we propose is the computation of atom sets. Computing the atom set of a Signal process, or determining that it cannot be computed, provides a proof of the fact that the process is, or is not, weakly endochronous (in the given stateless abstraction). As we shall see later in this paper, in cases where the process is not weakly endochronous, intuitive error messages can be produced with counterexamples showing the synchronization defects.

When a process is weakly endochronous and satisfies the input choice property, its atom set can be used to generate asynchronous or GALS implementations like those described in Section 3. A very simple way of producing such GALS components is to structure their GALS wrapper as the product of one AFSM per atom in the atom set. The AFSM associated with an atom cyclically performs the following sequence of steps:

1. Wait until:
   - All the inputs needed by are available on the input channels.
   - The synchronous process is not used by some other atom to compute its reaction. In the case where certain sub-processes can be directly activated by the wrapper (as explained in the end of Section 3.1), wait until none of the sub-processes needed by the computation of the atom are currently activated.
2. Activate the synchronous process (or the needed sub-processes) to compute the reaction of the atom.
3. When a synchronous process has completed the computation of its reaction, signal the availability of its results on the output channels.
4. After the output values are consumed:
   - Signal that the inputs of the process have been consumed, so that other inputs can arrive.
   - De-activate the synchronous process (or sub-processes).

This form of GALS wrapper can be implemented in both software and hardware. A similar hardware implementation is provided by Dasgupta *et al.* [9].

The GALS wrapper corresponding to the example in Fig. 8 will be the composition of 3 AFSMs corresponding to the 3 atoms. The AFSM associated with atom ($1^\bullet$ $1^\bullet$ $2^\bullet$ $2^\bullet$ 1^1 2^1 $\bullet$ $\_A$ $X^\bullet$) will wait until messages are available on channels 1 and 1, with the message on 1 carrying value 1. Then, the synchronous process of the first adder is triggered. When the first adder completes its execution, output 1 is signaled as available. Then, the wrapper waits until messages are available on 2 (with value 1) and 2, and then triggers the second adder. When this adder completes weakly endochronous programs for

e5.64 0 Td [(3.99878(a)3(e)4(p)5(p)5 15.001(e)14.9988(x

ossale (for programs that are not weakly endochronous).

5.3. Computing atom sets

For clarity reasons, we split the presentation of our analysis technique in two. This section deals with the variable renaming issues during composition of sub-processes. It also provides an algorithm for comput-

obtained by composing existing

e3.36 0 Td [(w)-3.99878(e)3.99878(a)3.99878(k)5(l)3(y)-292(e)3.99878(n)5(d)5(o)5(c)3.99878(h)5(r)3
e)3.99878(s)3.99878(s)3.99878]TJ /R29 10.9091 Tf 17.16 0 Td [(P)4.00146]TJ /R8 10.9091 Tf 12.12 0 Td [(i)3(s)-337.001(o)5(b)5(t)3(
(a)3.99878(m)-3(i)3(n)5(g)-380(()3(a)-370.001(r)3(e)3.99878(a)3.99878(c)3.99878(t)3(i)3.00244(o)5(n)5]TJ /R29 10.9091 Tf 165.84 0

$_i \leftarrow \quad _i \mid 1 \leq \quad \leq \quad]$) be the set of all the transformed reactions $[X_i \leftarrow \quad _i \mid 1 \leq \quad \leq \quad]$, where ranges over the atoms of A ( ) that are compatible with the renaming $[X \leftarrow$ $\leftarrow \quad \mid 1 \leq \quad \leq \quad]$) is the atom set generating the reactions of $\mathcal{R}($ $[X_i \leftarrow \quad _i \mid 1 \leq \quad \leq \quad])$.

Proof: Proving this consists (a)4n proving that:

$$_i \leftarrow \quad _i \mid 1 \leq \quad \leq \quad]) \text{ generates } \mathcal{R}(\quad [X_i \leftarrow \quad _i \mid 1 \leq \quad \leq \quad])$$

- No single element of A ($[X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ]$) can be generated by the other generators.

The first property is a simple consequence of the definitions of both $\mathcal{R}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ and the fact that A ( ) generates $\mathcal{R}(\ )$. The second property is determined by the minimality of A ( ). □

Recall now from Section 4.4 the definition of $\mathcal{R}^{\mathcal{V}(P)}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$. To generate its reactions we need:

- The atoms of A ($[X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ]$) with the support extended to $\mathcal{V}(\ )$ by padding with $\perp$ values.
- New atoms allowing us to generate any valuation for the newly-added signals, independently from the atoms obtained from . Such a set of atoms is given by A $(\mathcal{V}_Q)$, where $\mathcal{V}_Q = \mathcal{V}(\ ) \setminus \mathcal{V}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ is the set of newly-added signals.

We shall denote the resulting set of atoms:

$$A \quad ^{\mathcal{V}(P)}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ]) = \{\ |_{\mathcal{V}(P)}| \ \in A \quad (\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])\} \cup \{\ |_{\mathcal{V}(P)}| \ \in \quad A \quad (\mathcal{V}_Q)\}$$

Again, it is easy to prove that A $^{\mathcal{V}(P)}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ is the set of atoms generating $\mathcal{R}^{\mathcal{V}}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$.

A similar construction applies to sub-process , allowing the computation of A $^{\mathcal{V}(P)}(\ [\ _i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ which is the set of atoms generating $\mathcal{R}^{\mathcal{V}}(\ [\ _i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$.

Procedure 1 WEParallelComposition

Input: A_q A_r: atom set
Output: A: atom set
1: $A \leftarrow \emptyset$
2: **for all** $\in A_q$ **do**
3: *WEParallelCompositionAux*(,$\perp$,A_q,A_r,A)

Procedure 2 WEParallelCompositionAux

Input: $_1$, $_2$ reactions, A_1,A_2 atom sets
Reference-passed: A: reaction set
1: **if** $_2 \setminus \ _1 \neq \perp$ **then**
2: **for all** $\in A_1$ **do**
3: **if** $_1$ and $_2$ and $\wedge(\ _2 \setminus \ _1) \neq \perp$ **then**
4: **if** $_1 \vee \ = \ _2$ **then** $A \leftarrow A \cup \{\ _2\}$
5: **else** *WEParallelCompositionAux*($_1 \vee$, $_2$,A_1,A_2,A)
6: **else** return
7: **else** *WEParallelCompositionAux*($_2$, $_1$,A_2,A_1,A)

Finally, from A $^{\mathcal{V}(P)}(\ [X_i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ and A $^{\mathcal{V}(P)}(\ [\ _i \leftarrow \ _i \mid 1 \leq \ \leq \ ])$ we need to build the atom set generating $\mathcal{R}(\ )$. To do this, we rely on Procedure 1 and its auxiliary Procedure 2. The procedures build all the minimal reactions of $\mathcal{R}(\ )$ different from $\perp$ (*i.e.* the atoms) by

computing all the minimal sub-sets of atoms of A $^{\mathcal{V}(P)}(\ [X_i \leftarrow\ _i \mid 1 \leq\ \leq\ ])$ and respectively A $^{\mathcal{V}(P)}(\ [\ _i \leftarrow\ _i \mid 1 \leq\ \leq\ ])$ that give by union the same reaction (the computed atom).

The search for minimal sub-sets of atoms is a combinatorial process. It starts (in Procedure 1) with one atom of the first atom set. Then, a call to Procedure 2 will try to match the non-absent values of with atoms $_1$ $_k$ of the second set. When all the values of are matched, if $=\ _1 \vee\ \vee\ _k$, then we have found a minimal reaction of . If not, then $_1 \vee\ \vee\ _k$ is greater than , and we have to match the values of $(\ _1 \vee\ \vee\ _k) \setminus$ with atoms of the first set, etc. Alternately matching with atoms from the second set and the first set is done *only when necessary*, hence the obvious minimality of the result. It is also easy to prove that the resulting atom set generates $\mathcal{R}(\ )$.

5.4. Abstraction issues

Given that analysis and code generation is done using an abstraction of the original program means that certain precautions need to be taken.

The fact that in the Signal language the state variables are also signals implies that the stateless abstraction is precise with respect to weak endochrony: If the abstracted program is weakly endochronous, then the initial program is weakly endochronous, too. This implies that the stateful program or the implementation are deterministic (scheduling-independent) whenever the abstraction is proved weakly endochronous. The converse implication is not true: If a program is weakly endochronous, then its stateless abstraction is not necessarily weakly endochronous. This means that our technique may reject (stateful) weakly endochronous programs.

The data abstraction is more problematic. Indeed, a program being weakly endochronous does not imply that its finite stateless abstraction is weakly endochronous, and the other implication does not hold, either. From a practical point of view, this means that the programmer should include in its model finite abstracted versions (new signals) of all the infinite-type signals that are needed to make the difference between transitions (between atoms). This abstraction can either be done by hand, or automatically (for specific infinite types and application domains), or a combination of both.

6. Checking weak endochrony

According to Theorem 5.1, checking weak endochrony is determining when an atom set can be constructed for a given process. We follow this approach by determining for each process one minimal set of supplementary synchronizations (under the form of signal absence constraints) allowing the construction of a generator set with atom-like properties. Process is weakly endochronous *iff* the generators are free of forced absence constraints.

6.1. Signal absence constraints

For processes that are not weakly endochronous, the set of reactions $\mathcal{R}(\ )$ is not closed under the operations $\vee$, $\wedge$, $\setminus$ defined in the previous section, meaning that we cannot use generation properties to represent $\mathcal{R}(\ )$ in a *compact* fashion. This is due to the fact that the model does not allow the representation of *absence constraints*, which are needed in order to represent the *reaction to signal absence*.

To allow compact representation, we enrich the model with absence constraints under the form of *constrained absence* $\perp\!\!\!\perp$ signal values which are added to the domain of each signal. A reaction sets signal to $\perp\!\!\!\perp$ to represent the fact that upon union ($\vee$) the signal must remain absent. This new value represents the classical synchronizing absence of the synchronous model, which must be preserved at composition time. *However, we are not interested in fully reverting to a synchronous setting, but in preserving as few synchronizations as needed to allow deterministic asynchronous execution.*

We denote with $\mathcal{D}_S^{\perp\!\!\!\perp} = \mathcal{D}_S^{\perp} \cup \{\perp\!\!\!\perp\}$ the new domain with $\perp \leq \perp\!\!\!\perp$. The operators $\wedge$, $\vee$, and $\setminus$ are extended accordingly. We denote with $\mathcal{R}^{\perp\!\!\!\perp}$ the set of valuations of the signals over the extended domains. On $\mathcal{R}^{\perp\!\!\!\perp}$ we can extend the operators $\wedge$, $\vee$, $\setminus$, and . We define the operator $[] : \mathcal{R}^{\perp\!\!\!\perp} \to \mathcal{R}$ that removes absence constraints (replaces $\perp\!\!\!\perp$ values with $\perp$). We also define the converse transformation $\mathcal{R} \ni \mapsto \overline{\ } \in \mathcal{R}^{\perp\!\!\!\perp}$ that transforms all the $\perp$ values of a reaction into $\perp\!\!\!\perp$ values. We denote $\perp\!\!\!\perp = \overline{\perp}$ the reaction assigning $\perp\!\!\!\perp$ to all signals.

Consider now $\in \mathcal{R}^{\perp\!\!\!\perp}$ such that $[\ ] \leq \ '$ for some $' \in \mathcal{R}(\ )$. Then, we denote:

$$\textit{Constraints}_P(\ ) = \bigwedge \{\overline{\ } \mid \ \in \mathcal{R}(\ ) \wedge \ \leq \overline{\ }\}$$

The valuation $\textit{Constraints}_P(\ ) \in \mathcal{R}^{\perp\!\!\!\perp}$ is called the constraint associated with in . Whenever $\textit{Constraints}_P(\ )(\ ) \neq \perp$ we know that any reaction $' \in \mathcal{R}(\ )$ including has $'(\ ) = \textit{Constraints}_P(\ )(\ )$ (the value is constrained to either $\perp\!\!\!\perp$ or some non-absent value).

6.2. Generators

We define in this section the notion of *minimal fully constrained non-interfering set of generators* of a process , which is very similar to an atom set, except (1) it can be computed for any process and (2) it involves absence constraints. Such generator sets will represent for us compact representations of $\mathcal{R}(\ )$, and the basic objects in our weak endochrony check technique. The reactions of such a generator set can be seen as tiles that can be united (when disjoint) to generate all other reactions. Generators can also be compared with the prime implicants of a logic formula – they are reactions of smallest support that generate all other reactions.

Definition 2. (Generator set)
Let be a process. A set $\subseteq \mathcal{R}^{\perp\!\!\!\perp}$ of partial reactions such that $[\ ] \neq \perp$ for all $\in$ is a generator set of if $\mathcal{R}(\ ) = \{[\bigvee_{g \in K} \ ] \mid \subseteq \wedge \}$.

As we are building our generator sets incrementally, it is essential they preserve all the synchronization information of the process, including all absence constraints. Such generator sets are called fully constrained.

Definition 3. (Fully constrained generator set)
A generator set of process is called fully constrained if for all $\in$ we have $= \textit{Constraints}_P(\ )$.

Finally, we are looking for generator sets with atom-like exclusiveness properties.

Definition 4. (Non-interfering generator set)
A generator set of process is called non-interfering if for all $_1 \ _2 \in$ with $_1 \ _2$ and $[\ _1] \wedge [\ _2] \neq \perp$ we have $_1 = \ _2$.

Every Signal process has a fully constrained non-interfering generator set, obtained by replacing $\perp$ with $\perp\!\!\!\perp$ in all the reactions of $\mathcal{R}(\ )$. But using this representation amounts to reverting to the synchronous model, and not exploiting the concurrency of the process. We are therefore looking for least synchronized generator sets exhibiting minimal absence constraints.

Definition 5. (Less synchronized generator set)
Let be a process and $_1$, $_2$ two generator sets for . We say that $_1$ is less synchronized than $_2$, denoted $_1 \preceq\ _2$, if for all $_2 \in\ _2$ there exists $\subseteq\ _1$ with $\bigvee_{g\in K}\ \leq\ _2$ and $[\bigvee$

$$
\begin{aligned}
\mathcal{V}_{\mathtt{X\hat{}=Y}} &= \{(X^{v} \quad ^{w}) \mid \quad \in \mathcal{D}_X \quad \in \mathcal{D}_Y\} \cup \\
&\quad \{(\quad ^{v}) \mid \quad \in \mathcal{D}_W \quad \in \mathcal{V} \setminus \{X \quad\}\} \\
\mathcal{V}_{\mathtt{X\hat{}<Y}} &= \{(X^{v} \quad ^{w}) \mid \quad \in \mathcal{D}_X \cup \{\perp\!\!\!\perp\} \quad \in \mathcal{D}_Y\} \cup \\
&\quad \{(\quad ^{v}) \mid \quad \in \mathcal{D}_W \quad \in \mathcal{V} \setminus \{X \quad\}\}
\end{aligned}
$$

Figure 10. Minimal generator sets for clock equations over signal set $\mathcal{V}$ (they can be derived from the primitives)

Note that $_A$ determined above is unique with the given properties. It is also interesting to note that whenever is weakly endochronous, A is also a minimal non-interfering generator set for (but not necessarily fully-constrained). □

If a process is not weakly endochronous, then there may exist several minimal non-interfering generator sets. We provide here a technique allowing the construction of one such generator set. Our technique works inductively: We compute a minimal generator set for each statement in a bottom-up fashion following the syntax of the process. We shall denote with $_p$ the minimal non-interfering generator set built for statement . When, due to signal scoping, we need to explicitly include in the notation the set $\mathcal{V}$ of signals over which the reactions of are defined, we shall extend the notation to $^{\mathcal{V}}_{p}$. Fig. 9 and Fig. 10 give minimal non-interfering generator sets for primitives and clock equations.

In the remainder of the paper, when saying minimal generator set, we mean a minimal fully constrained non-interfering generator set.

6.3. Algorithms

6.3.1. Composition of atom sets

Our computation of minimal generator sets follows the same general pattern as the composition of atom sets of Section 5.3 (matching of minimal atom sets by combinational search). Therefore, we shall not repeat all the explanations of Section 5.3 but instead mark the differences with the new algorithms.

The composition of atom sets proceeds in a bottom-up fashion starting at low-level weakly endochronous processes which are then hierarchically composed to form larger ones. The computation of minimal generator sets also proceeds in a bottom-up fashion, but starts directly from the Signal statements, whose generator sets have been given in Figures 9 and 10.

Like in Section 5.3, the main difficulty is that of determining a minimal generator set for a composed statement from the minimal generator sets of the direct sub-statements. Without losing generality, we also consider here the case where two statements/processes (and) are composed. We assume all variable renaming has been already taken care of following the techniques of Section 5.3, and we provide here the equivalent of the Procedures 1 and 2 of Section 5.3.

The two driver procedures (*ParallelComposition* and *ParallelCompositionAux*) are in fact very similar (even syntactically) to Procedures 1 and 2. They perform basically the same combinatorial task: The exploration of all minimal combinations of generators in and whose present signals hold the same values. They operate by incrementally adding generators of and on one side in an attempt to match present values on the other side. The iteration stops when the generators match or when all possibilities have been exhausted without success.

Procedure 3 ParallelComposition

Input: $_q$ $_r$: generator set
Output: : generator set
1: $'\leftarrow\emptyset$
2: **for all** $\in$ $_q$ **do**
3: *ParallelCompositionAux*(,$\perp$, $_q$, $_r$, $'$)
4: $\leftarrow$ *MinimizeSynchronization*($'$)

However, operating on generator sets (as opposed to atom sets) largely complicates the task, as we shall see in the remainder of the section.

The first issue complicating the algorithms is that the matching process computes in variable $'$ of procedure *ParallelComposition* a generator set that may not be minimal with respect to $\preceq$. Intuitively, this is due to the fact that the composition of two minimal generator sets may render useless some of the absence constraints that were necessary in the individual components. When the resulting generator set is not minimal, procedure *MinimizeSynchronization* removes unnecessary synchronizations, as explained below.

Procedure 4 ParallelCompositionAux

Input: $_1$, $_2$ reactions, $_1$, $_2$ generator sets
Reference-passed: : reaction set
1: **if** $[\ _2]\setminus[\ _1]\neq\perp$ **then**
2: **for all** $\in$ $_1$ **do**
3: **if** $_1$ and $_2$ and $[\ ]\wedge([\ _2]\setminus[\ _1])\neq\perp$ **then**
4: **if** $[\ _1\vee\ ]=[\ _2]$ **then** $\leftarrow$ $\cup\{\ _1\vee\ _2\vee\ \}$
5: **else** *ParallelCompositionAux*($_1\vee$, $_2$, $_1$, $_2$,)
6: **else** return
7: **else** *ParallelCompositionAux*($_2$, $_1$, $_2$, $_1$,)

The `forall` loop in Procedure *ParallelComposition* determines a fully-constrained, non-interfering generator set for the composition of and . Formally, it computes all minimal (in the sense of inclusion) non-void non-contradictory subsets $\{\ ^q_1\ \ ^q_m\}\subseteq$ $_q$ and $\{\ ^r_1\ \ ^r_n\}\subseteq$ $_r$ with $[\ ^q_1\vee\ \ \vee\ ^q_m]=[\ ^r_1\vee\ \ \vee\ ^r_n]$ and $^q_1\vee\ \ \vee\ ^q_m\ \ ^r_1\vee\ \ \vee\ ^r_n$. For each such pair, it places in $'$ the generator $^q_1\vee\ \ \vee\ ^q_m\vee\ ^r_1\vee\ \ \vee\ ^r_n$. The differences between Procedure 2 (*WEParallelCompositionAux*) and the code of Procedure 4 emphasize the hybrid nature of $\perp\!\!\!\perp$: It forces a signal to be absent, but does not need to be matched in both and during our combinational search. This is why:

- The tests in lines 1, 3, and 4 work on reactions stripped of $\perp\!\!\!\perp$ values.
- The reaction added to in line 4 includes all the forced absence constraints of $_1$, $_2$, and (as opposed to just the present values, which are all present in $_2$).

The resulting generator set $'$ is not necessarily minimal. For instance, consider the statement $_0$ obtained by composing with no signal renaming $_0=$`(| C ^< B | C ^# A |)` with

$_0$ =(| C ^= when false |) (meaning $_0$ forces C to always be absent). The generator set computed for $_0$ by the forall loop is:

$$' = \{(A^\bullet\ B^{\perp\!\!\!\perp}\quad {}^{\perp\!\!\!\perp})\ (A^{\perp\!\!\!\perp}\ B^\bullet\quad {}^{\perp\!\!\!\perp})\ (A^\bullet\ B^\bullet\quad {}^{\perp\!\!\!\perp})\}$$

which is not minimal, as the minimal generator set (where A and B are independent) is:

$$_{P_0} = \{(A^\bullet\ B^\perp\quad {}^{\perp\!\!\!\perp})\ (A^\perp\ B^\bullet\quad {}^{\perp\!\!\!\perp})\}$$

The needed refinement of $'$ into $_{P_0}$ is done by Procedure *MinimizeSynchronization*, which is called by Procedure *ParallelComposition*. The procedure takes as input a non-interfering fully-constrained generator set and produces a minimal non-interfering fully constrained generator set. It works by uncovering concurrency by determining that existing generators can be further decomposed into less synchronized generators. To do so, the procedure attempts to remove one by one each forced absence value of each generator, and then uses Procedure *RemoveOneSynchronization* to obtain a fully constrained, non-interfering generator set where the chosen forced absence value is not necessary, and which is therefore less synchronized than the previous one.

Procedure 5 MinimizeSynchronization

Input: : set of generators over the set of signals $\mathcal{V}$
Reference-passed: $'$: set of generators over $\mathcal{V}$

1: **while** true **do**
2: Choose $\in$, $\in \mathcal{V}$ with ( ) $= {}^{\perp\!\!\!\perp}$ and
RemoveOneSynchronization( ) = ($''$)
for some $''$.
3: **if** there exist such , , and $''$ **then** $\leftarrow$ $''$
4: **else** $' \leftarrow$; return

The procedure terminates when no more $^{\perp\!\!\!\perp}$ values can be removed. When this happens, some $^{\perp\!\!\!\perp}$ values may remain in the generator set. Some of them, like those in our previous example ($_{P_0}$), are only there to ensure that the generator set is fully constrained.[6] When all remaining $^{\perp\!\!\!\perp}$ values are of this type, which is the case in our example, the program is weakly endochronous, and the atom set is obtained by removing all $^{\perp\!\!\!\perp}$ values from the generator set. Checking that this is the case amounts to checking that the set $\{[\ ]\ |\ \in\ _{P_0}\}$ satisfies the properties of an atom set.

When this is not the case, additional $^{\perp\!\!\!\perp}$ values are synchronization defects potentially leading to non-determinism upon execution in an asynchronous environment. For instance, the generator set of X^<Y (given in Fig. 10) contains such synchronization defects.

Procedure *MinimizeSynchronization* is a simple driver routine, the actual complexity of the synchronization minimization process being hidden within Function *RemoveOneSynchronization*. This function takes as input a generator set that has $^{\perp\!\!\!\perp}$ values in its generators, and the position of one of these values (given as the generator $_0$ and the signal $_0$ with $_0(\ _0) = {}^{\perp\!\!\!\perp}$). The output is *false* if this $^{\perp\!\!\!\perp}$ value cannot be removed without compromising overall synchronization. The output is *true* when the value can be removed. In this case, the function also returns a non-interfering fully constrained generator set $'$ that:

[6] Removing these $^{\perp\!\!\!\perp}$ values produces a set of reactions that is still a minimal generator set, although not fully constrained.

Procedure 6 RemoveOneSynchronization

Input: : set of generators over $\mathcal{V}$, $_0 \in$, $_0 \in \mathcal{V}$
such that $_0(\ _0) = \perp\!\!\!\perp$
Output: :Boolean
$'$: set of generators, if =

1: $^- \leftarrow \bigwedge\{\ ' \mid\ ' \in \ \wedge [\ _0] \leq\ '\}$
2: **if** $^-(\ _0) = \perp\!\!\!\perp$ **then**
3: $\leftarrow$
4: return
5: $'_0 \leftarrow\ _0$
6: $'_0[\ _0] \leftarrow \perp$
7: $' \leftarrow \{\ '_0\}$
8: $'' \leftarrow \{\ _0\}$
9: $\leftarrow\ \setminus \{\ _0\}$
10: **while** true **do**
11: Choose $_1 \in$, $'_1 \in\ '$ with $_1$ $'_1$ and $_1 \wedge\ '_1 \neq \perp$
12: **if** such a pair exists **then**
13: $\leftarrow\ \setminus \{\ _1\}$
14: $'' \leftarrow\ '' \cup \{\ _1\}$
15: $_{tmp} \leftarrow \{\ _1 \wedge\ ' \mid (\ ' \in\ ') \wedge (\ _1$ $') \wedge ([\ _1 \wedge\ '] \neq \perp)\}$
16: $_{tmp} \leftarrow\ _{tmp} \cup \{\ ' \setminus\ _1 \mid (\ ' \in\ ') \wedge (\ _1$ $') \wedge ([\ ' \setminus\ _1] \neq \perp)\}$
17: $_{tmp} \leftarrow\ _1 \setminus \bigcup_{g' \in G', g' \bowtie g_1} (\ _1 \wedge\ ')$
18: **if** $[\ _{tmp}] \neq \perp$ **then** $_{tmp} \leftarrow\ _{tmp} \cup \{\ _{tmp}\}$
19: $_{tmp} \leftarrow\ _{tmp} \cup \{\ ' \mid (\ ' \in\ ') \wedge (\ _1\ /\ ')\}$
20: $' \leftarrow\ _{tmp}$
21: **else**
22: $\leftarrow$ *CheckEquivalence*$(\ '\ '')$
23: $' \leftarrow\ ' \cup$
24: return

- Generates the same process as , in the sense of Definition 2.
- Is less synchronized than .
- Includes , with $\neq \emptyset$ and and $[\bigvee_{g \in K}\ ] = [\ _0]$ and $\bigvee_{g \in K}\ \leq\ _0[\ _0 \leftarrow \perp]$, where $_0[\ _0 \leftarrow \perp]$ is the reaction obtained from $_0$ by replacing the value of $_0$ with $\perp$.

The last condition means that $'$ no longer uses the synchronization represented by $_0(\ _0) = \perp\!\!\!\perp$ to preserve the global synchronous semantics.

The function works by removing the chosen $\perp\!\!\!\perp$ value from the generator set and then iteratively computing all the intersections and differences of non-contradictory reactions until no changes occur. Such overlapping non-contradictory generators cannot exist in the initial generator set, but the removal of the $\perp\!\!\!\perp$ value may introduce such cases. Computing the intersections and differences of non-contradictory

reactions is done assuming that the removed $\perp\!\!\!\perp$ value was covering the independence of smaller behaviors, like in our previous example (if we assume such a case, then the set of generators is closed under difference and intersection, much like the reaction sets of a weakly endochronous program). If the independence assumption is true, then the resulting set of reactions is a generator set generating the same process as . If not, the result is a non-interfering generator set that generates strictly more reactions than .

Compared with the intuitive description above, the algorithm is slightly optimized, in the sense where the resulting equivalence check is confined to the subset ′ of containing the generators that are replaced, and subset ″ containing their replacements (non-void minimal intersections and differences of generators in ′, after $_0(\ _0)$ has been assigned value $\perp$).

On the sub-sets ′ and ″, checking the generation equivalence (done by Function *CheckEquivalence*) consists in checking that ′ generates $\{[\ ]\mid\ \in\ ''\}$, in the sense of Definition 2 (because the elements in ″ are mutually contradictory).

For example, in the computation of $_{P_0}$ above, we can start by removing the $\perp\!\!\!\perp$ value of B in the first generator of ′. Then, function *RemoveOneSynchronization* will produce $_{P_0}$. No further simplification is possible.

Procedure 7 CheckEquivalence

Input: ′ ″: sets of generators, with ′ less synchronized than ″
Output: :Boolean
1: ″ ← the set of reactions generated by ′
2: ← $\{[\ ]\mid\ \in\ \} = \{[\ ]\mid\ \in\ ''\}$

Once a minimal generator set is computed for a program , we use the criterion of Theorem 6.1 to check whether is weakly endochronous or not. When is not weakly endochronous, intuitive error messages can be provided, showing each pair of generators that are interferent, yet can only be distinguished using a forced absent value.

Using the previous algorithms to compute the minimal fully constrained non-interfering generator set of the process in Fig. 7 gives:

$$\{(\ 1^\bullet\ \ 1^\bullet\ \ \perp\!\!\!\perp)\ (\ 2^\bullet\ \ 2^\bullet\ \ \perp\!\!\!\perp)$$
$$(\ 1^\bullet\ \ 1^\bullet\ \ 2^\bullet\ \ 2^\bullet\ \ ^\bullet\ \ ^\bullet)\}$$

As expected, the process is not weakly endochronous because $[(\ 1^\bullet\ \ 1^\bullet\ \ 2^\bullet,\ \ 2^\bullet\ \ ^\bullet)]$ and $[(\ 1^\bullet\ \ 1^\bullet\ \ \perp\!\!\!\perp)]$ are neither conflicting, nor of disjoint support. We have already provided, in Section 5.1, the result of the computation (the set of atoms) for the weakly endochronous process of Fig. 8.

6.3.2. Variable scoping

The previous algorithms have been defined under the subprocess instantiation rules of Section 4.4, which transform the local signals of a sub-process into local signals of the instantiating process itself. But the process instantiation rule can also be used to hide the internal signals of instantiated processes. There are two potential advantages in doing so: Obviously, further analysis is simplified because there are fewer signals.

Less obviously, the restriction may reveal potential concurrency. Internal signals of a synchronous process can in fact be seen as internal computing resources. Whenever two computations/atoms use the same signal,[7] they cannot be both active during a given execution instant, even if their sets of inputs, outputs, and internal state elements are completely disjoint (so they are functionally independent). Forgetting how the computations are actually implemented, and looking only at their interface footprint may then reveal that they are intrinsically independent, and can be re-implemented in this way, with a different set of internal signals.

From a practical point of view, when the goal is the generation of concurrent (multi-task) implementations, the hiding of internal signals should be used with much care. Indeed, hiding local signals can hide actual dependencies and create the false impression that reactions are non-interferent when in fact they are interfering in the sub-process.

When the goal is the generation of sequential (single-task) code or the verification of certain properties, hiding variables will simplify the interface and the representation of the process. We provide here a routine allowing the hiding of local signals.

Procedure *SignalScope* builds the generator set of a process in which is the set of private signals. The input of the function is the set of generators, as computed by the algorithms of the previous section (with the signals of visible). The output is a set of generators for over signal set $\mathcal{V}( ) \setminus$. This means that signals of can no longer be used to differentiate between non-independent generators.

In the definition of our procedure, we use a scoping (hiding) operator defined as follows: If is a reaction over the set of signals $\mathcal{V}$ and $\subseteq \mathcal{V}$, then $\setminus$ denotes the restriction $|_{\mathcal{V}\setminus V}$ of on $\mathcal{V} \setminus$. We also denote with $\perp\!\!\!\perp_V$ the reaction over $\mathcal{V}$ that equals $\perp\!\!\!\perp$ on and $\perp$ elsewhere.

Procedure 8 SignalScope

Input: : set of generators over $\mathcal{V}$
$\subset \mathcal{V}$: set of signals
Output: $'$:set of generators over $\mathcal{V} \setminus$

$\leftarrow \{ \in \mid [\setminus] \neq \perp\}$
while true **do**
  Choose $_1$ $_2 \in$ with $(_1 \setminus) \setminus (_2 \setminus) \neq \perp$ and $_1 \setminus$ $_2 \setminus$ and $(_1) \cap$ $(_2) \neq \emptyset$
  if there exist such $_1$, $_2$ **then**
    Choose $\in$ $((_1 \setminus) \setminus (_2 \setminus))$
    $' \leftarrow$ $_2$
    $'[ ] \leftarrow \perp\!\!\!\perp$
    $'' \leftarrow \{ \in \setminus \{ _2\} \mid (_2) \wedge (/ ')\}$
    $''' \leftarrow \{ _2 \vee \perp\!\!\!\perp_{supp(g)} \vee \perp\!\!\!\perp_{supp(g_2)} \vee _2 \mid \in ''\}$
    $\leftarrow (\setminus (\{ _2\} \cup ')) \cup '''$
  else
    $' \leftarrow \{ \setminus \mid \in \}$
  return

Our procedure works by finding pairs of generators that are not independent, yet are not distinguishable in an asynchronous environment using only the signals of $\mathcal{V} \setminus$. This is done by the `while` loop

[7] Or the same set of signals defining a shared resource, such as an adder.

in line 2 and the choice in line 3. Whenever such a pair $(\ _1\ \ _2)$ is discovered, one absence constraint is added to $_2$ to allow distinction, and possibly other generators are affected (because the added absence constraint may render non-independent generators that were previously independent).

6.3.3. Complexity

While the theoretical complexity of our problem (and especially the size of the intermediate representation) may seem prohibitive, it is usually the case that in real-life applications, due to the regularity of human-made designs, these theoretical bounds are never reached, and computations are tractable. We are currently developing more efficient symbolic representations for generator sets and more efficient analysis routines, possibly borrowing from existing clock calculus techniques already used in Signal/Polychrony and other synchronous languages.

7. Conclusion

We have defined a general method to characterize and synthesize semantics-preserving wrappers to execute synchronous processes on a globally asynchronous architecture. This method considers processes abstracted by high-level synchronization constraints and is thus applicable to a large variety of scenarios. Although we chose the Signal language to illustrate our approach, the method itself is independent of a domain-specific formalism.

GALS architectures constructed with our method have a predictable behavior that is sound and complete with respect to initial synchronous specifications, regardless of the size of the system or of latency in the network. The result of the analysis allows to directly synthesize executives for all specifications whose processes are proven stateless weakly endochronous. Moreover, in the case a specification fails to meet expected criteria, our analysis points directly at the faulty synchronization issue(s).

In the present paper, our main concern was to characterize an effective criterion ensuring the functional correctness of GALS architectures in an untimed setting. A longer-term objective is to take real-time requirements into account. This should provide guarantees on more elaborate constraints pertaining to periodicity, throughput, WCET.

Such an extension requires the definition of timing analysis and scheduling techniques compatible with our program execution model. Yet, the executives themselves could be simplified under specific timing hypothesis (for instance, a FIFO protocol can be simplified if the reader is faster than the writer, etc.). In parallel, we are also investigating ways to optimize the representation of atoms by using decision trees.

References

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

[2] Benveniste, A., Caillaud, B., Guernic, P. L.: Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation, *Information and Computation*, **163**, 2000, 125 – 171.

[3] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., de Simone, R.: The Synchronous Languages 12 Years Later, *Proceedings of the IEEE*, **91**(1), January 2003, 64–83.

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

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

[6] Caspi, P., Girault, A., Pilaud, D.: Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors, *IEEE Transactions on Software Engineering*, **25**(3), May/June 1999, 416–427.

[7] Chapiro, D. M.: *Globally asynchronous, locally synchronous systems*, Ph.D. Thesis, Stanford University, Department of Computer Science, 1984.

[8] Cortadella, J., Kondratyev, A., Lavagno, L., Sotiriou, C.: Desynchronization: Synthesis of Asynchronous Circuits from Synchronous Specifications, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25**(10), October 2006, 1904–1921.

[9] Dasgupta, S., Potop-Butucaru, D., Caillaud, B., Yakovlev, A.: Moving from Weakly Endochronous Systems to Delay-Insensitive Circuits, *Electronic Notes in Theoretical Computer Science*, **146**(2), 2006, 81 – 103, Proceedings of the Second Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS 2005).

[10] Grandpierre, T., Sorel, Y.: From Algorithm and Architecture Specification to Automatic Generation of Distributed Real-Time Executives: a Seamless Flow of Graphs Transformations, *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

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

[12] Kahn, G.: The semantics of a simple language for parallel programming, *Information Processing '74* (J. Rosenfeld, Ed.), North Holland, 1974.

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

[14] Mazurkiewicz, A.: *Concurrent Program Schemes and their Interpretations*, Technical report, DAIMI, Arhus University, 1977.

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

[16] Nebut, M.: Specification and Analysis of Synchronous Reactions, *Formal Aspects of Computing*, **16**(3), august 2004, 263–291.

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

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

[19] Potop-Butucaru, D., de Simone, R., Sorel, Y.: Necessary and sufficient conditions for deterministic desynchronization, *Proceedings EMSOFT'07*, Vienna, Austria, 2007.

[20] Potop-Butucaru, D., de Simone, R., Sorel, Y.: *deterministic execution of synchronous programs in an asynchronous environment. A compositional necessary and sufficient condition*, Research Report RR-6656, INRIA, 2008.

[21] Singh, M., Theobald, M.: Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures, *Proceedings DATE'04*, Paris, France, 2004.

[22] Vijayaraghavan, M., Arvind: Bounded Dataflow Networks and Latency-Insensitive Circuits, *Proceedings Memocode'09*, Nice, France, 2009.