From Concurrent Multiclock Programs to Deterministic Asynchronous Implementations

Dumitru Potop-Butucaru Robert de Simone Yves Sorel Jean-Pierre Talpin
INRIA, France
{FirstName.LastName}@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. Based on the theory of weakly endochronous systems, our technique uses a compact representation of the abstract synchronization configurations of the analyzed process to determine a minimal set of synchronization patterns generating all possible reactions.

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 multiclocked 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 [11]. 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: Section 2 and Section 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 [11] and adapts it to our framework. Section 6 defines novel algorithms to determine if a specification is weakly endochronous. We conclude in Section 7.

2 Multiclock synchronous system

We use a small, intuitive example to present our problem, the desired result, and the main implementation is-

sues. 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 C wire whenever SYNC is present.

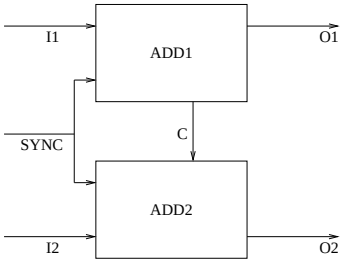

Figure 1. Data-flow of a configurable adder. [1]

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 $\mathcal{V}(P)$ the finite set of signals of a process P. We shall distinguish inside $\mathcal{V}(P)$ the disjoint sub-sets of *input* and *output signals*, respectively denoted $\mathcal{I}(P)$ and $\mathcal{O}(P)$.

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

If we denote with EXAMPLE our configurable adder, then

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

All signals are typed. We denote with $\mathcal{D}_{\mathsf{S}}$ 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 $\mathcal{D}_{\mathsf{S}}$. Otherwise, we say that it is *absent*. Absence is simply represented with value $\perp$, which is appended to all domains $\mathcal{D}_{\mathsf{S}}^{\perp} = \mathcal{D}_{\mathsf{S}} \cup \{\perp\}$. Formally, a reaction of the process is a valuation of its signals into their extended domains $\mathcal{D}_{\mathsf{S}}^{\perp}$. 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 Fig. 2 is $\{I1, I2, O1, O2\}$. In a reaction r, we distinguish the *input event*, which is the restriction $r\mid_{\mathcal{I}(\texttt{EXAMPLE})}$ of r to input signals, and the *output event*, which is the restriction $r\mid_{\mathcal{O}(\texttt{EXAMPLE})}$ 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 [6] uses a dedicated `event` type of domain $\mathcal{D}_{\texttt{event}} = \{\bullet\}$. We follow the same convention: In our example, SYNC has type `event`. To represent reactions, we use a *set-like convention* and omit signals with value $\perp$. In Fig. 2, the signal types are $SYNC : event$, $O1, O2 : integer$, $I1, I2 : integer\_pair$, $C : Boolean$. Reaction 4 is denoted $(I1^{(9,9)}, O1^{8}, I2^{(0,0)}, O2^{0})$. The *stuttering reaction* assigning $\perp$ to all signals is denoted $\perp$. Reaction 5 is a stuttering reaction.

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.

We assume that the execution of the synchronous process is a cyclic repetition of 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.

In order to achieve deterministic execution,[2] the main difficulty lies in step (1), as it involves the potential recon-

[1] 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.

[2] Like in [10], 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 no changes in the following technical results, we preferred a simpler presentation.

struction 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. 3, 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 3. 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 • on SYNC. Depending on the arrival order, which cannot be determined, any of the reactions $(I1^{(1,2)}, O1^3, SYNC^\bullet, C^0)$ or $(I1^{(1,2)}, O1^3)$ can be executed, 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).

We have seen in the previous section that deterministic input event reconstruction is impossible for some synchronous processes. This means that a methodology to implement synchronous processes on an asynchronous architecture 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 a random synchronous process can be done in two steps:

1. transforming the initial process, through added synchronizations and/or signals, so that it belongs to the implementable class, and then
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. This paper proposes a technique for checking weak endochrony of real-life (real-size) specifications.

3.1 Previous work. Motivation

The most developed notions identifying classes of implementable processes are the concepts of *latency-insensitive systems* of Carloni *et al.* [4] and the *endochronous systems* of Benveniste *et al.* [2, 6]. 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.

The *endochronous systems* and the related hardware-centric *generalized latency-insensitive systems* [14] 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. 4 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 4. Endochronous solution

Checking endochrony consists in ordering the signals of the process in a tree representing the incremental presence inference process (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 subprocesses 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 over-synchronized.

Weak endochrony, due to Potop, Caillaud, and Benveniste [11] and presented in Section 5, generalizes endochrony by allowing both synchronized and non-synchronized (independent) computations to be realized by a given process.

Fig. 5 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.

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).

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

Figure 5. Weakly endochronous solution

4 Multiclock Specification in Signal

The use of weakly endochronous processes allows the preservation of the independence of non-synchronized computations,[3] 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 clock synchronization constraints we are interested in. Like other synchronous data-flow formalisms, such as Lustre, Scade, 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 [9]. 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 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 [13] can be easily preserved in this way.

However, the abstraction means that: (1) Certain weakly endochronous processes are rejected, as the analysis cannot determine it and (2) The code generated for a weakly endochronous process may be over-synchronized.

4.2 Process structure

In Signal, a specification is a *process*, whose definition may involve other processes, hierarchically. Fig. 6 gives the Signal process corresponding to the configurable adder of

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

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).

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

Figure 6. The Signal process of the configurable adder in Fig. 1

The data-flow 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. The hierarchy of processes allows the structuring of a specification and the definition of signal scopes that mask internal signals. Process EXAMPLE using process ADD1 in its data-flow intuitively corresponds to replacing each instance of ADD1 in EXAMPLE with its data-flow with the internal signals of ADD1 being masked.

4.3 Data-flow

The data-flow 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. The use of a constraint language allows us to easily manipulate functionally incomplete specifications.

4.3.1 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*. Lines 8 and 9 of Fig. 6, 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. Clock exclusion is not used in our example. Writing "S1^#S2" requires that 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 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 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. 7, "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.

5 Weak endochrony

The theory of *weakly endochronous (WE) systems* [11], gives criteria establishing that a synchronous presentation hides a behavior that is fundamentally asynchronous and

[4] X is defined by V the first time Y occurs and then takes the previous value of Y

deterministic. Absence information is not needed, which guarantees the deterministic implementability of the synchronous specification in an asynchronous environment.[5]

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).

```
process EXAMPLE2 = (? boolean S1, S2;
                      event I1,I2
                    ! event O1,O2 )
(| (O1,O2) := EXAMPLE (when S1, I1, I2)
 | when S1 ^= when S2
 |)
where
 process EXAMPLE = the process in Fig. 6
end
```

Figure 7. A weakly endochronous refinement of process EXAMPLE is obtained by limiting the use of signal absence (when compared to the other solutions)

This property suggests a natural organization of the possible values of a signal S as a Scott domain defined by $\perp \leq v$, for all $v \in \mathcal{D}_S$. The domain structure on particular signals induces a product partial order $\leq$ on reactions with $r_1 \leq r_2$ 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.

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 $r_1 \bowtie r_2$, both $r_1 \vee r_2$ and $r_1 \wedge r_2$ are defined, and we can also define the difference $r_1 \backslash r_2$, which has support $supp(r_1) \backslash supp(r_2)$ and equals r_1 on its support. For a set K with $\bowtie$ we denote $\vee K = \bigvee_{r \in K} r$.

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

[5]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 [10], the axioms of weak endochrony can be traced down to the fundamental result of Keller [7] 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 [8].

Definition 1 (stateless weak endochrony) *We say that process P is weakly endochronous its set of reactions $\mathcal{R}(P)$ is closed under the operations associated to the previously-defined domain structure: intersection, union, and difference of non-contradictory reactions.*

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 P, denoted $Atoms(P)$ is the set of the smallest (in the sense of $\leq$) reactions of $\mathcal{R}(P)$ different from $\perp$. The set of atomic transitions is characterized by two fundamental properties: non-interference and generation.

Theorem 1 (atom set characterization) *A stateless process P is weakly endochronous if and only if there exists a set of reactions $A \subseteq \mathcal{R}(P)$ such that:*

Generation: *The union of non-interfering atoms generates all the reactions of $\mathcal{R}(P)$: $\mathcal{R}(P) = \{\bigvee K \mid K \subseteq G\wedge \bowtie K\}$.*

Non interference: *Two distinct atoms $a_1, a_2 \in A$, $a_1 \neq a_2$ are either contradictory or independent.*

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 1 corresponds to the case where no distinction is made between input, output and internal signals of a system (which is the case in [11]). 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.[6] Formally:

Input choice: For any two contradictory atoms $a_1, a_2 \in A$, there exists $s \in \mathcal{V}(P)$ such that $a_i(s) \neq \perp, i = 1, 2$, and $a_1(s) \neq a_2(s)$.

6 Checking weak endochrony

According to Theorem 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 P 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 P is weakly endochronous *iff* the generators are free of forced absence constraints.

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

6.1 Signal absence constraints

For processes P that are not weakly endochronous, the set of reactions $\mathcal{R}(P)$ is not closed under the operations $\vee$, $\wedge$, $\setminus$ defined in the previous section, meaning that we cannot use generation properties to represent $\mathcal{R}(P)$ 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. An extended reaction r sets signal S to $\perp\!\!\!\perp$ to represent the fact that upon union ($\vee$) the signal S 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 $\bowtie$. 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 $\overline{\ } : \mathcal{R} \to \mathcal{R}^{\perp\!\!\!\perp}$ 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.

6.2 Generators

We define in this section the notion of *minimal fully constrained non-interfering set of generators* of a process P, which is very similar to an atom set, except (1) it can be computed for any process P and (2) it involves absence constraints. Such generator sets will represent for us compact representations of $\mathcal{R}(P)$, 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 P be a process. A set $G \subseteq \mathcal{R}^{\perp\!\!\!\perp}$ of partial reactions such that $[g] \neq \perp$ for all $g \in G$ is a generator set of $\mathcal{R}(P)$ if $\mathcal{R}(P) = \{[\bigvee_{g\in K} g] \mid K \subseteq G \wedge \bowtie K\}$.*

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 G of process P is called fully constrained if each atom represents all absence constraints associated to it. Formally, for all $g \in G$ we have: $g = \bigwedge\{\overline{r} \mid r \in \mathcal{R}(P) \wedge g \leq r\}$.*

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

Definition 4 (Non-interfering generator set) *A generator set G of process P is called non-interfering if for all $r_1, r_2 \in G$ with $r_1 \bowtie r_2$ and $[r_1] \wedge [r_2] \neq \perp$ we have $r_1 = r_2$.*

Every Signal process has a fully constrained non-interfering generator set, obtained by replacing $\perp$ with $\perp\!\!\!\perp$ in all the reactions of $\mathcal{R}(P)$. 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 P be a process and G_1, G_2 two generator sets for P. We say that G_1 is less synchronized than G_2, denoted $G_1 \preceq G_2$, if for all $g_2 \in G_2$ there exists $K \subseteq G_1$ with $\bigvee_{g\in K} g \leq g_2$ and $[\bigvee_{g\in K} g] = [g_2]$.*

The procedures of the next section will build for each process a fully constrained, non-interfering generator set that is minimal in the sense of $\preceq$.

Theorem 2 *Let P be a process and G be a fully constrained, non-interfering generator set that is minimal in the sense of $\preceq$. Then, G is weakly endochronous if and only if the set $A_G = \{[g] \mid g \in G\}$ satisfies the generation and non-interference properties of Theorem 1.*

Proof sketch: If A_G satisfies the given property, then according to Theorem 1 we know that P is weakly endochronous.

In the other sense, assume A is the set of atoms of the weakly endochronous process P. For all $a \in A$ we define

$$g_a = \bigwedge\{\overline{r} \mid r \in \mathcal{R}(P) \wedge a \leq r\}$$

Then, $G_A = \{g_a \mid a \in A\}$ is a fully constrained non-interfering generator set that can be proved minimal in the sense of $\preceq$ (ant it is unique with this property). $\square$

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

$$G^{\mathcal{V}}_{\texttt{X:=Y default Z}} =$$

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

$$G^{\mathcal{V}}_{\texttt{X:=Y when Z}} =$$

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

$$G^{\mathcal{V}}_{\texttt{X:=f(Y}_1\texttt{,...,Y}_n\texttt{)}} =$$

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

Figure 8. Minimal generator sets for primitive Signal equations

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

Figure 9. Minimal generator sets for clock equations (can be derived from primitives)

shall denote with G_p the minimal non-interfering generator set built for statement p. When, due to signal scoping, we need to explicitly include in the notation the set $\mathcal{V}$ of signals over which the reactions of p are defined, we shall extend the notation to $G^{\mathcal{V}}_p$. Fig. 8 and Fig. 9 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

Given a Signal program, the computation of a minimal generator set proceeds bottom-up in the syntax tree, starting from the minimal generator sets of the primitives (given above), and incrementally computing one minimal generator set for each composed statement. Composed statements are of only two types: parallel composition and sumodule instantiation. This section deals with parallel composition: The main algorithm is *ParallelComposition*, which computes a minimal generator set of $p \mid q$ starting from minimal generator sets of p and q.

The signal scoping realized by subprocess instantiation must be ignored, meaning that the local signals of the generators of the sub-process are treated as local signals of the instantiating process itself. This is necessary if the goal is multi-task code generation, because hiding local signals can hide actual dependencies and render non-interfering reactions that are interfering in the sub-process. For space reasons, we relegate to Appendix A the routine allowing the hiding of local signals at the level of subprocess instantiation (which may be useful for verification purposes).

Function 1 ParallelComposition

Input: G_p, G_q: generator set
Output: $G_{p|q}$: generator set
  $G' \leftarrow \emptyset$
  for all $g \in G_p$ **do**
    ParallelCompositionAux(g,$\perp$,G_p,G_q,G')
  $G_{p|q} \leftarrow$ *MinimizeSynchronization*(G')

Function *ParallelComposition* and the recursive auxiliary (Procedure 2) perform an exploration of all combinations of generators in p and q whose present signals hold the same values. It operates by incrementally adding atoms of p and q 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.

Procedure 2 ParallelCompositionAux

Input: r_1,r_2 reactions, G_1,G_2 generator sets
Reference-passed: G: 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 *ParallelCompositionAux*($r_1 \vee g$,r_2,G_1,G_2,G)
  else *ParallelCompositionAux*(r_2,r_1,G_2,G_1,G)

The `forall` loop in Function *ParallelComposition* determines a fully-constrained, non-interfering generator set for $p \mid q$, but which is not necessarily minimal. For instance, consider the parallel composition $p_0 \mid q_0$, where p_0 is `(| C ^< B | C ^# A |)` and q_0 is `(| C ^= when false |)` (meaning q_0 forces `C` to always be absent). The generator set computed for $p_0 \mid q_0$ by the `forall` loop is:

$$G' = \{(A^\bullet, B^{\perp\!\!\!\perp}, C^{\perp\!\!\!\perp}), (A^{\perp\!\!\!\perp}, B^\bullet, C^{\perp\!\!\!\perp}), (A^\bullet, B^\bullet, C^{\perp\!\!\!\perp})\}$$

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

$$G_{p_0|q_0} = \{(A^\bullet, B^\perp, C^{\perp\!\!\!\perp}), (A^\perp, B^\bullet, C^{\perp\!\!\!\perp})\}$$

The needed decomposition of G' into $G_{p_0|q_0}$ is done by Procedure *MinimizeSynchronization.* The procedure uncovers concurrency by determining that existing generators can be further decomposed into less synchronized generators. It works by attempting to remove one by one each forced absence value of each generator, and then using Function *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 3 MinimizeSynchronization

Input: G: set of generators over the set of variables $\mathcal{V}$
Reference-passed: G': set of generators over $\mathcal{V}$
while true **do**
Choose $g \in G$, $s \in \mathcal{V}$ with $g(s) = \perp\!\!\!\perp$ and *RemoveOneSynchronization*$(G, g, s) = (true, G'')$ for some G''.
if there exist such g, s, and G'' **then** $G \leftarrow G''$
else $G' \leftarrow G$; 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 ($G_{p_0|q_0}$), are only there to ensure completeness with respect to Definition 3, and can be safely removed upon execution in an asynchronous environment. 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 for each generator g that:

$$g = \bigwedge\{g' \mid g' \in G \wedge [g] \leq g'\}$$

This means that the generator contains no $\perp\!\!\!\perp$ value in addition to those prescribed by Definition 3.

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. 9) contains such synchronization defects.

The complexity of the *MinimizeSynchronization* procedure is hidden within the auxiliary Function *RemoveOneSynchronization.* After removal of one synchronization ($\perp\!\!\!\perp$ value), this function successively computes all intersections and differences of non-contradictory generators until no changes occur. This process, which may remove further $\perp\!\!\!\perp$ values, results in a less synchronized generator set satisfying the non-interfering and fully constrained properties. But we still need to check that it still is a generator set for the considered process (that the removal of synchronizations does not allow supplementary behaviors).

This check is done by Function *CheckEquivalence.* When *CheckEquivalence* returns false, we know that the particular $\perp\!\!\!\perp$ value given as input to Function *RemoveOneSynchronization* cannot be removed (it is needed to preserve the synchronous semantics). For example, in the computation of $G_{p_0|q_0}$ above, we can assume that we start by removing the $\perp\!\!\!\perp$ value of B in the first generator of G'. Then, Function *RemoveOneSynchronization* will produce $G_{p_0|q_0}$. No further simplification is possible.

Function 4 RemoveOneSynchronization

Input: G: set of generators over $\mathcal{V}$, $g_0 \in G$, $s_0 \in \mathcal{V}$ such that $g_0(s_0) = \perp\!\!\!\perp$
Output: $Status$:Boolean
G': set of generators, if $Status = true$
$\overline{g} \leftarrow \bigwedge\{g' \mid g' \in G \wedge [g] \leq g'\}$
if $\overline{g}(s_0) = \perp\!\!\!\perp$ **then**
$Status \leftarrow false$
return
$g'_0 \leftarrow g_0$
$g'_0[s_0] \leftarrow \perp$
$G' \leftarrow \{g'_0\}$
$G'' \leftarrow \{g_0\}$
$G \leftarrow G \setminus \{g_0\}$
while true **do**
Choose $g_1 \in G$, $g'_1 \in G'$ with $g_1 \bowtie g'_1$ and $g_1 \wedge g'_1 \neq \perp$
if such a pair exists **then**
$G \leftarrow G \setminus \{g_1\}$
$G'' \leftarrow G'' \cup \{g_1\}$
$G_{tmp} \leftarrow \{g_1 \wedge g' \mid (g' \in G') \wedge (g_1 \bowtie g') \wedge ([g_1 \wedge g'] \neq \perp)\}$
$G_{tmp} \leftarrow G_{tmp} \cup \{g' \setminus g_1 \mid (g' \in G') \wedge (g_1 \bowtie g') \wedge ([g' \setminus g_1] \neq \perp)\}$
$g_{tmp} \leftarrow g_1 \setminus \bigcup_{g' \in G', g' \bowtie g_1} (g_1 \wedge g')$
if $[g_{tmp}] \neq \perp$ **then** $G_{tmp} \leftarrow G_{tmp} \cup \{g_{tmp}\}$
$G_{tmp} \leftarrow G_{tmp} \cup \{g' \mid (g' \in G') \wedge (g_1 \not\bowtie g')\}$
$G' \leftarrow G_{tmp}$
else
$Status \leftarrow$ *CheckEquivalence*(G', G'')
$G' \leftarrow G' \cup G$
return

The computation of the intersections and differences between generators, and the equivalence check are optimized in Function *RemoveOneSynchronization* to only involve generators that are actually modified. Non-interfering atoms are not changed or analyzed.

Using the previous algorithms to compute the minimal

Function 5 CheckEquivalence

Input: G', G'': sets of generators, with G' less synchronized than G''

Output: $Equiv$:Boolean

$G'' \leftarrow$ the set of reactions generated by G'

$Equiv \leftarrow \{[g] \mid g \in G\} = \{[r] \mid r \in G''\}$

fully constrained non-interfering generator set of the process in Fig. 6 gives:

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

As expected, the process is not weakly endochronous because $[(I1^\bullet, O1^\bullet, I2^\bullet, O2^\bullet, SYNC^\bullet)]$ and $[(I1^\bullet, O1^\bullet, SYNC^{\perp\!\!\!\perp})]$ are neither conflicting, nor of disjoint support. The minimal non-conflicting generator set of the transformed process in Fig. 7 is:

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

The process is weakly endochronous.

7 Conclusion

We have defined a general method to characterize and synthesize correctness-preserving wrappers to execute synchronous processes on a globally asynchronous architecture. This methods 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 protocols 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 better using, e.g., decision trees.

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] 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.

[5] 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.

[6] 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.

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

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

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

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

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

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

[13] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

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

A Appendix

Function *SignalScope* builds the generators set of a process P in which V is the set of private signals. The input of the function is the set of generators of the program body. The output is the set of generators of P, which don't use variables of V to differentiate between non-independent generators. In its definition, we use the scoping (hiding) operator: If r is a reaction over the set of variables $\mathcal{V}$ and $V \subseteq \mathcal{V}$, then $r \setminus V$ denotes the reaction over $\mathcal{V} \setminus V$ that equals r on variables of $\mathcal{V} \setminus V$. We also denote with $\perp\!\!\!\perp_V$ the reaction over $\mathcal{V}$ that equals $\perp\!\!\!\perp$ on V and $\bot$ elsewhere. The function works by adding supplementary synchronizations (constrained absent values), when necessary.

Function 6 SignalScope

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

$G \leftarrow \{g \in G \mid [g \setminus V] \neq \bot\}$
while true **do**
  Choose $g, g' \in G$ with $(g \setminus V) \setminus (g' \setminus V) \neq \bot$ and $g \setminus V \bowtie g' \setminus V$ and $(g \setminus V) \wedge (g' \setminus V) \neq \bot$
  if there exist such g, g' **then**
    Choose $s \in supp((g \setminus V) \setminus (g' \setminus V))$
    $g'' \leftarrow g'$
    $g''[s] \leftarrow \perp\!\!\!\perp$
    $G'' \leftarrow \{g \in G \setminus \{g'\} \mid (g \bowtie g') \wedge (g \not\bowtie g'')\}$
    $G''' \leftarrow \{g' \vee \perp\!\!\!\perp_{supp(g)}, g \vee \perp\!\!\!\perp_{supp(g')}, g \vee g' \mid g \in G''\}$
    $G \leftarrow (G \setminus (\{g'\} \cup G'')) \cup G'''$
  else
    $G' \leftarrow \{g \setminus V \mid g \in G\}$
    return