From Concurrent Multi-clock 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 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 [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. 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 C wire whenever SYNC is present.

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.
Fig. 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 \( \mathcal{V}(P) \) 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 \( \mathcal{O}(P) \).

![Diagram](image)

Fig. 2. A synchronous run of the adder

If we denote with \( \text{EXAMPLE} \) our configurable adder, then

\[
\begin{align*}
\mathcal{V}(\text{EXAMPLE}) &= \{I1, I2, \text{SYNC}, O1, O2, C\} \\
\mathcal{I}(\text{EXAMPLE}) &= \{I1, I2, \text{SYNC}\} \\
\mathcal{O}(\text{EXAMPLE}) &= \{O1, O2\}
\end{align*}
\]

All signals are typed. We denote with \( D_S \) the domain of a signal \( S \). Not all signals need to have a value in a reaction, to model cases where only parts of the process compute. We will say that a signal is present in a reaction when it has a value in \( D_S \). Otherwise, we say that it is absent. Absence is simply represented with value \( \perp \), which is appended to all domains \( D_S^\perp = D_S \cup \{\perp\} \). Formally, a reaction of the process is a valuation of its signals into their extended domains \( D_S^\perp \). We denote with \( \mathcal{R} \) the set of all such valuations. The support of a reaction \( r \), denoted \( \text{supp}(r) \), 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 \( r \mid \mathcal{I}(\text{EXAMPLE}) \) of \( r \) to input signals, and the output event, which is the restriction \( r \mid \mathcal{O}(\text{EXAMPLE}) \) 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 \( D_{\text{event}} = \{\bullet\} \). We follow the same convention:

In our example, \( \text{SYNC} \) has type \( \text{event} \). To represent reactions, we use a set-like convention and omit signals with value \( \perp \). In Fig. 2, the signal types are \( \text{SYNC} : \text{event}, O1, O2 : \text{integer}, I1, I2 : \text{integer_pair}, C : \text{Boolean} \). Reaction 4 is denoted \( (I1^{(9,9)}, O1^{5}, O2^{(0,0)}) \).

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.

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 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. 3, that we consider the inputs of Fig. 2 without synchronization information.

\[
\begin{array}{ccccccc}
\text{Clock} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{I1} & (1,2) & \perp & (9,9) & (9,9) & \perp & (2,5) & \perp \\
\text{O1} & 3 & \perp & 8 & 8 & \perp & 7 & \perp \\
\text{SYNC} & \perp & \perp & \bullet & \perp & \perp & \bullet & \perp \\
\text{C} & \perp & \perp & 1 & \perp & \perp & 0 & \perp \\
\text{I2} & \perp & \perp & (0,0) & (0,0) & \perp & (1,4) & (2,3) \\
\text{O2} & \perp & \perp & 1 & 0 & \perp & 5 & 5 \\
\end{array}
\]

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 \text{ADD1} will then receive the first value \((1,2)\) on the input channel \text{I1} and \(\bullet\) on \text{SYNC}. Depending on the arrival order, which cannot be determined, any of the reactions \((I1^{(1,2)}, O1^{3}, \text{SYNC}^{\bullet}, C^{0})\) or \((I1^{(1,2)}, O1^{5})\) 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 another.

\(2\) 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.
the other (so one can’t deterministically choose between
them in an asynchronous environment).

Earlier we have seen that deterministic input event
reconstruction is impossible for some synchronous pro-
cesses. This means that a methodology to implement syn-
chronous 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 implementa-
tion to a random synchronous process can be done in two steps:

1) transforming the initial process, through added syn-
chronizations 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 en-
dochronous 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 im-
plement, 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.

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].

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>⊥</td>
</tr>
<tr>
<td>ADD1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>⊥</td>
<td>1</td>
<td>⊥</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>I2</td>
<td>⊥</td>
<td>(0.0)</td>
<td>(0.0)</td>
<td>(1.4)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>O2</td>
<td>⊥</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4. Endochronous solution

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
over-synchronized.

Weak endochrony, due to Potop, Caillaud, and Ben-
veniste [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 exec-
ution 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 exec-
ution 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 indepen-
dently elsewhere (between synchronization points).

4. Multi-clock Specification in Signal

The use of weakly endochronous processes allows the
preservation of the independence of non-synchronized
computations,3 while adding the supplementary synchro-
nization needed to ensure deterministic execution in an

3. So that later analysis or implementation steps can exploit it.
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\(^4\) and (2) The code generated for a weakly endochronous process may be oversynchronized.

\(^4\) 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.

### 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 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 reaction of \(P\). We denote with \(R(P)\) 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 \(^\wedge S\), 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 ^= 01” specifies that signal I1 is present in a reaction iff 01 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, ^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$ in t V” \footnote{5}, 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 \textit{when C}” is equal to X whenever the boolean signal C is present with value true. Otherwise, it is \perp. The shortcut for “\textit{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 \textit{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 \cite{11}, 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.\footnote{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 \textit{don’t care value} imposing no synchronization constraint (as opposed to present values).

\begin{verbatim}
process EXAMPLE2 = (? boolean S1, S2;
  event I1, I2
  ! event O1, O2 )
  | (O1, O2) := EXAMPLE (when S1, I1, I2)
  when S1 \textasciitilde when S2
end

process EXAMPLE = the process in Fig. 6 end
\end{verbatim}

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 \( v \in D_S \). The domain structure on particular signals induces a product partial order \leq on reactions with \( r_1 \leq r_2 \) if and only if \( \text{supp}(r_1) \subseteq \text{supp}(r_2) \) and \( r_1(v) = r_2(v) \) for all \( v \in \text{supp}(r_1) \).

We say of two reactions \( r_1 \) and \( r_2 \) that they are non-contradictory, written \( r_1 \Join r_2 \), if \( r_1(v) = r_2(v) \) for all \( v \in \text{supp}(r_1) \cap \text{supp}(r_2) \). Otherwise, we say that the reactions are contradictory, written \( r_1 \not\Join r_2 \). Given a set of reactions \( K \), we shall say that it is non-contradictory, denoted \( \Join K \) 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 \( \lor \) and \( \land \), and called union and intersection of reactions. If \( r_1 \Join r_2 \), both \( r_1 \lor r_2 \) and \( r_1 \land r_2 \) are defined, and we can also define the difference \( r_1 \setminus r_2 \), which has support \( \text{supp}(r_1) \setminus \text{supp}(r_2) \) and equals \( r_1 \) on its support. For a set \( K \) with \( \Join K \) we denote \( \lor r \in K r \).

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

\begin{definition}[stateless weak endochrony] We say that process \( P \) is weakly endochronous if its set of reactions \( R(P) \) is closed under the operations associated to the previously-defined domain structure: intersection, union, and difference of non-contradictory reactions.
\end{definition}

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

\begin{theorem}[atom set characterization] A process \( P \) is weakly endochronous if and only if there exists a set of reactions \( A \subseteq R(P) \) such that:
\end{theorem}
• **Generation**: The union of non-interfering atoms generates all the reactions of $\mathcal{R}(P)$: $\mathcal{R}(P) = \{ \vee K \mid K \subseteq A \wedge \bowtie K \}$.
• **Non interference**: Two distinct atoms $a_1, a_2 \in A$, $a_1 \neq a_2$ 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 [11])

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. 7 Formally:

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

6.1. 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 $\bot$ signal values which are added to the domain of each signal. An extended reaction $r$ sets signal $S$ to $\bot$ 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.

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 $G \subseteq \mathcal{R}(P)$ of partial reactions such that $g \neq \bot$ for all $g \in G$ is a generator set of $\mathcal{R}(P)$ if $\mathcal{R}(P) = \{ \bigwedge_{g \in G} g \mid K \subseteq G \bowtie K \}$.

As we are building our generator sets incrementally, it is essential that 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: $g = \bigwedge \{ r \mid r \in \mathcal{R}(P) \wedge g \leq r \}$.

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 $[r_1] \wedge [r_2] \neq \bot$ we have $r_1 \approx r_2$.

Every Signal process has a fully constrained non-interfering generator set, obtained by replacing $\bot$ with $\bot$ 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, $P$ 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. Moreover, $P$ is deterministic if and only if $G$ satisfies the input choice property of Section 5.

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

$$g_a = \bigwedge \{T \mid r \in R(P) \land a \preceq r\}$$

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 $\preceq$ (and it is unique with this property). □

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 $G_p$ 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 $V$ of signals over which the reactions of $p$ are defined, we shall extend the notation to $G_p^V$. 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.

$$G^V_{K=\emptyset} = \{ (X^v, Y^v, Z^w) \mid v \in D_X, w \in D_X \cup \{\bot\} \} \cup \{ (X^v, Y^\bot, Z^w) \mid v \in D_X \} \cup \{ (W^v) \mid v \in D_W, W \in V \setminus \{X, Y, Z\}\}$$

$$G^V_{K=G} = \{ (X^v, Y^w) \mid v \in D_X, w \in D_Y \} \cup \{ (W^v) \mid v \in D_W, W \in V \setminus \{X, Y\}\}$$

$$G^V_{K=X} = \{ (X^v, Y^w) \mid v \in D_X \cup \{\bot\}, w \in D_Y \} \cup \{ (W^v) \mid v \in D_W, W \in V \setminus \{X, Y\}\}$$

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

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 submodule 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 actually interfering in the sub-process.

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

$ParallelCompositionAux(g, \bot, G_p, G_q, G')$

$G_p|q \leftarrow \text{MinimizeSynchronization}(G')$

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. Our function 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. When the resulting generator set is not minimal, function $MinimizeSynchronization$ removes unnecessary synchronizations, as explained below.

The forall loop in Function $ParallelComposition$ determines a fully-constrained, non-interfering generator set for $p \mid q$. Formally, it computes all minimal (in the sense of inclusion) non-void non-contradictory subsets $\{g_1' \cup \ldots \cup g_n'\} \subseteq G_p$ and $\{g_1'' \cup \ldots \cup g_m''\} \subseteq G_q$ with $\bigvee g_1'' \cup \ldots \cup g_m'' = \bigvee g_1' \cup \ldots \cup g_n'$ and $g_1'' \cup \ldots \cup g_m'' \not\preceq g_1' \cup \ldots \cup g_n'$ and places in $G'$ the generators $g_1' \cup \ldots \cup g_n'$, or $g_1'' \cup \ldots \cup g_m''$.

The resulting generator set $G'$ is not necessarily minimal. For instance, consider the parallel composition $p_0 |
Procedure 2 ParallelCompositionAux

Input: $r_1, r_2$ reactions, $G_1, G_2$ generator sets
Reference-passed: $G$: reaction set

if $[r_2] \setminus [r_1] \neq \bot$ then
  for all $g \in G_1$ do
    if $g \bowtie r_1$ and $g \bowtie r_2$ and $[g] \land (\{r_2\} \setminus \{r_1\}) \neq \bot$ then
      if $[r_1 \lor g] = [r_2]$ then $G \leftarrow G \cup \{r_1 \lor r_2 \lor g\}$
    else ParallelCompositionAux($r_1 \lor g, r_2, G_1, G_2, G$)
  end for
else ParallelCompositionAux($r_2, r_1, G_2, G_1, G$)

$q_0$, where $p_0$ is $(| C \bowtie B | C \bowtie A |)$ and $q_0$ is $(| C \bowtie \text{when false} |)$ (meaning $q_0$ forces $C$ to always be absent). The generator set computed for $p_0 \mid q_0$ by the forall loop is:

$$G' = \{(A^\bot, B^\bot, C^\bot), (A^\bot, B^\bot, C^\bot), (A^\bot, B^\bot, C^\bot)\}$$

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

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

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 $g(s) = \bot$ 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 $\bot$ values can be removed. When this happens, some $\bot$ 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 $\bot$ 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 $\bot$ values from the generator set. Checking that this is the case amounts to checking for each generator $g$ that:

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

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

When this is not the case, additional $\bot$ 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.

Function 4 RemoveOneSynchronization

Input: $G$: set of generators over $\mathcal{V}$, $g_0 \in G$, $s_0 \in \mathcal{V}$ such that $q_0(s_0) = \bot$
Output: Status:Boolean

$G'$: set of generators, if $Status = true$

$\exists i \{g' \mid g' \in G \land [g] \leq g'\}$

if $\exists j \{g \mid g \in G \land [g] \leq g\}$

Choose $g_1 \in G$, $g_1' \in G'$ with $g_1 \bowtie g_1'$ and $g_1 \land g_1' \neq \bot$

if such a pair exists then
  $G' \leftarrow G' \setminus \{g_1\}$
  $G'' \leftarrow G'' \cup \{g_1\}$
  $G_{tmp} \leftarrow \{g_1 \land g_1' \mid (g_1' \in G') \land (g_1 \bowtie g_1') \land ([g_1 \land g_1'] \neq \bot)\}$
  $G_{tmp} \leftarrow G_{tmp} \cup \{g_1 \land (g_1' \land (g_1 \bowtie g_1') \land ([g_1 \land g_1'] \neq \bot))\}$
  $G_{tmp} \leftarrow G_{tmp} \cup \{g_1 \land (g_1' \land (g_1 \bowtie g_1') \land ([g_1 \land g_1'] \neq \bot))\}$
  $G' \leftarrow G' \setminus \{g_{tmp}\}$
else
  $Status \leftarrow \text{CheckEquivalence}(G', G'')$
  $G' \leftarrow G' \cup G$
  return

The complexity of the MinimizeSynchronization procedure is hidden within the auxiliary Function RemoveOneSynchronization. Formally, this function takes a set of reactions, iteratively computes all the intersections and differences of non-contradictory reactions until no changes occur. The result of the process is a set of reactions $G'$ satisfying the non-interference property of Theorem 1, generating a set of reactions that includes the reactions generated by $G$, and such that the number of all the forced absence values of all the reactions of $G'$ is strictly smaller than the similar number for $G$. We preserve $G''$ generator set for $p \mid q$ that is less synchronized than $G$ whenever $G''$ generates the same reactions as $G$ (that the removal of synchronizations does not allow supplementary behaviors).

8. The result of removing of a forced absence value from a generator of a generator set is not necessarily a generator set.
This check is done by Function CheckEquivalence. When CheckEquivalence returns false, we know that the particular \(\bot\) 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 \(\bot\) value of \(B\) in the first generator of \(G'\). Then, Function RemoveOneSynchronization will produce \(G_{p_0|q_0}\). No further simplification is possible.

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.

**Function 5 CheckEquivalence**

**Input:** \(G', G''\): sets of generators, with \(G'\) less synchronized than \(G''\)

**Output:** Equiv: Boolean

\[G'' \leftarrow \{g | g \in G'\} \quad \text{Equiv} \leftarrow \{(r | r \in G'')\}\]

Once a minimal generator set is computed for a program \(P\), we use the criterion of Theorem 2 to check whether \(P\) is weakly endochronous or not. When \(P\) 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. 6 gives:

\[\{ (I_1^*, O_1^*, SYNC\uparrow), (I_2^*, O_2^*, SYNC\downarrow), \\
(I_1^*, O_1^*, I_2^*, O_2^*, SYNC^*, C^*) \}\]

As expected, the process is not weakly endochronous because \(\{(I_1^*, O_1^*, I_2^*, O_2^*, SYNC^*)\}\) and \(\{(I_1^*, O_1^*, SYNC\downarrow)\}\) are neither conflicting, nor of disjoint support. The minimal non-conflicting generator set of the transformed process in Fig. 7 is:

\[\{ (I_1^*, O_1^*, SYNC\uparrow, C\downarrow, SYNC\downarrow), \\
(I_2^*, O_2^*, SYNC\downarrow, C\downarrow, SYNC\downarrow), \\
(I_1^*, O_1^*, I_2^*, O_2^*, SYNC^1, SYNC^2, C^*), SYNC^* \}\]

The process is weakly endochronous.

**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 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**


