The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded System
Abdoulaye Gamatié, Thierry Gautier

To cite this version:

HAL Id: hal-00550056
https://hal.archives-ouvertes.fr/hal-00550056
Submitted on 23 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
The Signal Synchronous Multi-clock Approach to the Design of Distributed Embedded Systems

Abdoulaye Gamatié and Thierry Gautier

Abstract—This paper presents the design of distributed embedded systems using the synchronous multi-clock model of the SIGNAL language. It proposes a methodology that ensures a correct-by-construction functional implementation of these systems from high-level models. It shows the capability of the synchronous approach to apply formal techniques and tools that guarantee the reliability of the designed systems. Such a capability is necessary and highly worthy when dealing with safety-critical systems. The proposed methodology is demonstrated through a case study consisting of a simple avionic application, which aims to pragmatically help the reader to understand the manipulated formal concepts, and to apply them easily in order to solve system correctness issues encountered in practice. The application functionality is first modeled as well as its distribution on a generic hardware architecture. This relies on the endochrony and endo-isochrony properties of SIGNAL specifications, defined previously. The considered architectures include asynchronous communication mechanisms, which are also modeled in SIGNAL and proved to achieve message exchanges correctly. Furthermore, the synchronizability of the different parts in the resulting system is addressed after its deployment on a specific execution platform with multi-rate clocks. After all these steps, a distributed code can be automatically generated.

Index Terms—Distributed embedded systems, Correct-by-construction design methodology, Safety-critical domains, Formal validation, Synchronous model, Asynchronous mechanisms, Multi-clock, SIGNAL language

1 INTRODUCTION

There are various reasons to distribute embedded systems: the high performance enabled by the use of several computation units for a better response time, sometimes the geographical delocalization of the system elements, the replication of systems for fault tolerance, etc. Typical domains in which distributed embedded systems are useful are the following:

- transportation: the last generation of cars and aircraft increasingly combine computers and electrical linkages. This combination makes comfortable the control of the engine. This combination leads to so-called fly-by-wire systems, e.g., adopted in Toyota Prius automobile or in the A380.
- telecommunication: cell phone manufacturers and Internet highly face problems related to distributed and mobile embedded systems deployed on networks. A major issue is the compatibility between heterogeneous devices, and the robust and transparent communications in networks, while guaranteeing performance and real-time properties.
- industrial automation: it covers industrial robots and embedded mechatronic systems, which comprise technologies from engineering disciplines in computer software and hardware as well as in mechanics and automatic control. The issues concern the design of distributed architectures for scalable and reconfigurable mechatronic systems, and of embedded control.

The design of distributed embedded systems generally has to take into account several crucial aspects: the used communication mechanisms must be proved to be sufficiently robust, i.e. no loss of messages; for sub-systems located on different sites, the duration of communications must not alter the real-time properties of the overall system, the global functionality must be guaranteed, etc. So, the distribution of a system has to follow well-defined methodologies that offer the suitable means to clearly express its inherent concurrency and to validate its behavior w.r.t. functional and non-functional requirements.

In the current study, we consider the synchronous model for design and validation. This model offers an adequate formal basis to deal with the reliable design of embedded systems. Its basic assumption is that computation and communications are instantaneous from the point of view of a logical time, referred to as "synchrony hypothesis". This favors deterministic models of system behaviors for safe analysis.

1.1 The synchronous multi-clock model of SIGNAL

The design of distributed systems has been extensively studied for decades [44], [19]. The asynchrony inherent to these systems appears a priori as an obstacle to their description with the synchronous model. According to [45], a fully synchronous system is characterized by the boundness and knowledge of: i) processing speed, ii) message delivery delay, iii) local clock rate drift, iv) load pattern, and v) difference among local clocks. A fully asynchronous system assumes none of these characteristics.

Distributed embedded systems are generally implemented with GALS architectures, which are suitably abstracted with the SIGNAL synchronous multi-clock or polychronous model [37]. The multi-clock model is gaining an increasing attention from academia and industry to address the modern electronic designs [13], [24]. According to the polychronous model, the activation clock of a system consists of the set of instants at which its components react. This set is only partially ordered since concurrent components may have their own (independent) activation clocks. When these components do not interact, it is not necessary that their clocks be synchronized. The central question is not to agree on a global time, but to agree on the common event occurrences during the interaction of components. Contrarily to the usual monolithic approach adopted by the other synchronous languages (e.g. LUSTRE, ESTEREL until recently), which consists of compact single-clocked designs, the
polychronous approach aims at designing a modular fashion
systems with multiple loosely coupled clocks so as to be able
to deal with the complexity of their distribution.

Compared to the synchrony/asyncrhony definition of [45],
the polychronous model offers an intermediate vision: while
it assumes the boundness of computation and communication
activities, the difference among local activation clocks is a priori
unknown. This vision is opposed to the monolithic one, which
considers the existence of a unique common clock in a system,
represented by a totally ordered set of instants. The SIGNAL
language relies on the polychronous model.

Fig. 1 illustrates the design philosophy behind polychrony:
to give an adequate abstract model of a system shown in
Fig. 1(b) which enables a subsequent choice towards a dis-
tributed system implemented by globally asynchronous locally
synchronous (GALS) architectures [17] shown in Fig. 1(a) or
a single-clocked system implemented by centralized architec-
tures shown in Fig. 1(c).

The polychronous system, depicted in Fig. 1(b), consists of
two concurrent sub-systems that receive control from events
a and b and conditionally communicate via an event c or do
something else (d or e). It can be refined into a single-clocked
system, Fig. 1(c), by synchronizing the events a and b. It can be
also refined into a distributed system, Fig. 1(a), by distribut-
ing a and b on two different locations and by implementing c
by a FIFO communication channel. In general, the reliability
of such channels is implicitly assumed in the polychronous
model. However, in this paper, it is explicitly proved.

1.2 Synchronous design of distributed systems

Among the large spectrum of literature devoted to the design
of distributed systems [14], [19], we mainly concentrate below
on related works that rely on the synchronous model.

As mentioned above, a major advantage of using such a
model for the development of distributed embedded systems
is that, on the one hand, the global correctness of systems
can be trust-worthily guaranteed using the associated formal
technology. It is particularly necessary when the system is used
in a safety-critical domain such as automotive or avionics.
This explains the numerous studies on program distribution
problematics in synchronous languages. Most of these studies
focus on automatic program distribution methods [27]: given
a centralized synchronous program P and a distributed archi-
tecture A, the deployment of P on A is defined automatically,
with the necessary inserted communication code, so that the
resulting distributed program has the same functional behavior
as P. However, note that beyond these studies, automatic
program distribution has been widely investigated [52].

The major contributions on this topic in the synchronous
approach community could be summarized according to the main
synchronous languages [10]: ESTEREL, LUSTRE and SIGNAL.

In ESTEREL, we can mention the work of Berry and Sentovich
[12] on the construction of GALS systems as synchronous cir-
cuits represented by a network of communicating codesign finite
state machines. GALS architectures [17] consist of components
that execute synchronously and communicate asynchronously.
Another relevant work concerns the ESTEREL specification and
programming of large-scale distributed real-time systems [33].

The LUSTRE language has been also used in several studies
on the design of distributed systems. Girault [26] addressed the
distribution of synchronous automata within the framework of
this language. Afterwards, he focused on further issues such as
automatic deduction of GALS systems from centralised synchronous
ar
cuits [28], and the optimized execution of desynchronized embedded reactive programs to guarantee
time-constraints [29]. We must note the very important
achievements by Caspi and his colleagues on the same topics
(see [27]). They recently proposed an approach to deploy
LUSTRE programs on time-triggered architectures [16].

In SIGNAL, there have been lots of results on program distrib-
ution, mostly from the theoretical viewpoint. The earlier work
of Chéron [18] dealt with the communication of separately
compiled SIGNAL programs. Then, Maffeis [39] showed how
to abstract such programs into graphs in order to define the
qualitative scheduling and partitioning of these graphs. The
work of Aubry [6] focused on similar problems as Girault [26]
by exploring the manual and semi-automatic distribution of
synchronous dataflow programs in SIGNAL. While these stud-
ies were mostly devoted to the practical side, Benveniste
and Le Guernic lead several theoretical works on the distribution
of SIGNAL programs [8], [9], [25], [11], [37], [41]. The approach
presented in this paper shows how these theoretical results are
exploited in practice to define correct-by-construction models
of distributed embedded systems.

Finally, we can notice other interesting contributions such as
[30] in which Grandpierre showed, with Petri nets, how one
can derive a distributed implementation from a synchronous
dataflow specification, e.g. in LUSTRE or SIGNAL, of a given
application while minimizing the response time w.r.t. real-time
requirements. The SYNDEx environment [31] aims at providing
designers with such program distribution facilities.

1.3 Our contribution: a seamless design methodology

While the above studies mainly concern automatic program
distribution, in this paper we focus on how to design entirely
distributed embedded systems with a synchronous language
by proposing a general methodology, Fig. 2, composed of four
steps detailed in the following chronological order:

1) System specification and manual distribution: modeling of
application functionality, distributed hardware architecture,
and their association. This step is similar to the
initial step in usual hardware/software codesign;

2) Automatic transformations: guaranteeing the global func-
tional correctness of the distribution. These transforma-
tions exploit the endochrony and endo-isochrony proper-
ties of SIGNAL programs, presented in Section 2.2;

3) Deployment on specific platforms: instantiating the parts
of the resulting model with components that represent
specific platform mechanisms, e.g., for communication, synchronization. There are only few existing works devoted to the synchronous modeling of asynchronous mechanisms [24], [35], [23].

4) Non functional analysis and automatic code generation: checking the non functional (temporal) constraints induced by the chosen deployment and distributed code generation.

The above methodology is exposed through a case study. It is built on several complementary results obtained previously. The second step of this methodology uses an implementation of the theoretical results of [8], [9]. The third step relies on the insights gained from [21], [23] in which we have proposed a library of components consisting of synchronous models of asynchronous mechanisms for communication, synchronization or execution. Here, we consider the mechanism defined in [21] to describe the communications in the deployed system model of our case study. We also give the proof of queueing order preservation in this mechanism, which was an assumption previously. On the other hand, while the design approach presented in [23] mainly targets integrated modular avionics architectures, the current methodology aims at more general multi-processor architectures and fully makes use of the SIGNAL multi-clocked model. The fourth step uses the non functional analysis techniques proposed in [36], [38] and implements code generation approach defined in [25].

1.4 Example: a Flight Warning System

We consider a running example from the avionic domain, consisting of a Flight Warning System (FWS) to illustrate our methodology. This system is used in the Airbus A340 aircraft. It has been proposed by the Aerospatiale Company (France) as a case study in [38]. The FWS system is in charge of deciding on when and how to emit warning signals whenever there is an anomaly during the operational mode of an airplane. It is illustrated in Fig. 3 together with its implementation architecture. It consists of two cyclic concurrent processes:

- given an alarm $a_i$, the alarm manager process confirms $a_i$ after a given period of time or removes $a_i$ from the set of confirmed alarms depending on the fact that $a_i$ is detected “present” or “absent”;
- the alarm notifier process emits warning signals associated with confirmed alarms.

In this paper, the exact way alarms are made present or absent is supposed to be out of scope. This is achieved in another part of the aircraft.

Even though the above two processes interact, they execute independently. So, their activation clocks are a priori not strongly correlated. Such a system is particularly well-suited to be described and analyzed with the polychronous model. The specific choice of executing the processes separately on different processors is one possibility among others. Here, it allows us to illustrate the possible design issues resulting from the distribution of a system functionality and how to address them with our proposed methodology.

1.5 Outline

The remainder of this article is organized as follows: Section 2 introduces the synchronous language SIGNAL and its associated multi-clock semantic model. Then, the next sections present our methodology in detail through the design of FWS on a distributed platform. Section 3 is devoted to the first and second steps. It addresses the SIGNAL description of the FWS system, its associated hardware architecture and its correct distribution on the chosen architecture. Section 4 concentrates on the third step. It deals with the modeling of an asynchronous communication mechanism, represented by a FIFO queue that is proved to be correct w.r.t. its expected services. This mechanism is used for the deployment of the FWS system on a specific platform. Section 5 focuses on the last step of our methodology. It addresses clock synchronizability issues in the
deployed model of the FWS system and the automatic code generation. Finally, our proposition is discussed in Section 6 and concluding remarks are given in Section 7.

2 The Signal Language

In order to define the formal semantics of SIGNAL, we introduce the polychronous semantic model in Section 2.1. Then, we give an overview of the language in Section 2.2.

2.1 The polychronous semantic model

We consider the following sets: \( X \) is a countable set of variables; \( B = \{f, \text{tt}\} \) is a set of Boolean values where \( f \) and \( \text{tt} \) respectively denote false and true; \( V \) is the domain of operands (at least \( V \supseteq B \)); and \( T \) is a dense set equipped with a partial order relation noted \( \leq \), and with a greatest lower bound. The elements of \( T \) are called tags.

We now introduce the notion of observation point.

Definition 1 (observation points): A set of observation points is a set of tags \( T \) such that: i) \( T \subset T; \) ii) \( T \) is countable; iii) each pair of tags admits a lower bound in \( T \).

The set \( T \) provides a discrete time dimension that corresponds to the logical instants at which the presence/absence of events can be observed during a system execution. The set \( \mathbb{T} \) provides a continuous (physical) time dimension. So, the mapping of \( T \) on \( \mathbb{T} \) allows one to move from “abstract” descriptions to “concrete” descriptions in the semantic model.

A chain \( C \subseteq T \) is a totally ordered set which admits a lower bound. The set of chains is denoted by \( C \). For a set of observation points \( T \), we denote by \( C_T \) the set of all chains in \( T \). The notations \( \text{min}(C) \) and \( \text{predec}(t) \) respectively mean the minimum of \( C \) and the immediate predecessor of a tag \( t \in C \).

Definition 2 (events, signals and behaviors, Fig. 4).

- An event \( e \) on a given set of observation points \( T \) is a couple \((t, v) \in T \times V \).
- A signal on a given set of observation points \( T \) is a partial function \( s \in T \rightarrow V \), associating values with observation points that belong to a chain \( C \in C_T \). The set of signals on \( T \) is noted \( \mathcal{S}_T \). The domain of \( s \) is denoted by \( \text{tags}(s) \).
- For a given set of observation points \( T \), a behavior \( b \) on \( X \subseteq X \) is a function \( b \in X \rightarrow \mathcal{S}_T \) that associates each variable \( x \in X \) with a signal \( s \) on \( T \). We denote by \( B_{T,X} \) the set of behaviors of domain \( X \subseteq X \) on \( T \). The set \( B_{T,X} \) represents the set that contains all the behaviors defined on the union of all the sets of variables on \( T \). Finally, we write \( \text{vars}(b) \) and \( \text{tags}(b) = \bigcup_{x \in \text{vars}(b)} \text{tags}(b(x)) \) to respectively denote the domain of \( b \) and its associated set of tags.

![Fig. 4. Behavior, events and signals.](image)

For any behavior \( b \) defined on \( X \subseteq X \), we note \( b_{\mathcal{X}} \), its projection on a set of variables \( X' \subseteq X \). In other words, \( \text{vars}(b_{\mathcal{X}}) = X' \) and \( \forall x \in X, b_{\mathcal{X}}(x) = b(x) \). The projection of \( b \) on the complement of \( X' \) in \( X \) is denoted by \( b_{\overline{X}} \).

A signal can be seen, in fact, as an elastic with ordered marks (its tags). If such an elastic is stretched, the marks remain in the same order but new marks may be inserted between stretched marks. If the elastic gets unstretched, the marks become closer to each other, but they still remain in the same order. The same holds for a set of elastics, i.e., a behavior.

Definition 3 (stretching): For a given set of observation points \( T \), a behavior \( b_1 \) is less stretched than a behavior \( b_2 \), noted \( b_1 \preceq b_2 \), iff there exists a bijection \( f : \text{tags}(b_1) \rightarrow \text{tags}(b_2) \) following which \( b_1 \) and \( b_2 \) are isomorph:

\[
\forall x \in \text{vars}(b_1), f(\text{tags}(b_1(x))) = \text{tags}(b_2(x)),
\]

\[
\forall x \in \text{vars}(b_1) \forall t \in \text{tags}(b_1(x)), b_1(x)(t) = b_2(x)(f(t)),
\]

\[
\forall t_1, t_2 \in \text{tags}(b_1), t_2 \leq t_2 \Rightarrow f(t_1) \leq f(t_2),
\]

and such that \( VC \in C_T, \ \forall t \in C \ t \leq f(t). \)

The stretching relation is a partial order on \( B_T \). It induces an equivalence relation between behaviors.

Definition 4 (stretch-equivalence): For a given set of observation points \( T \), two behaviors \( b_1 \) and \( b_2 \) are stretch-equivalent, noted \( b_1 \equiv b_2 \), iff there exists a behavior \( b_3 \) less stretched than \( b_1 \) and \( b_2 \), i.e. \( b_1 \equiv b_2 \) iff \( \exists b_3 \ b_1 \preceq b_3 \) and \( b_3 \preceq b_2 \).

The class of equivalence of a behavior following the stretch-relation forms a semi-lattice, which admits a minimal element. We call strict behaviors those which are minimal for the stretch-relation on \( T \). For a given behavior \( b \), the set of all behaviors that are stretch-equivalent to \( b \) on \( T \) defines its stretch-closure on \( T \), noted \( b^* \).

Definition 5 (stretch-closure of a behavior set): The stretch-closure of a set of behaviors \( p \) on a given set \( T \) of observation points is the set \( p^* \), which includes all behaviors resulting from the stretch-closure of each behavior \( x \in p, i.e. \ p \subseteq p^* = \bigcup_{b \in p} b^* \).

The stretch-closure allows us to define the following process notion, which is a specific set of behaviors:

Definition 6 (process): For a given set of observation points \( T \), a process \( p \) is a stretch-closed set of behaviors, i.e. \( p \in \mathcal{P}(B) \) such that \( p = p^* \).

We write \( \text{vars}(p) \) to denote the set of variables of a process \( p \) (we say equivalently that \( p \) is defined on \( \text{vars}(p) \)). Every non empty process contains a subset \( p_1 \subseteq p \) of strict behaviors (for each \( b_1 \in p_1 \), there exists a unique \( b_2 \in (p_1) \) such that \( b_1 \preceq b_2 \)).

The notions presented in this section are sufficient to express the semantics of SIGNAL basic concepts within the polychronous model [37]. In the remainder of the paper, we also use them to characterize some model properties.

2.2 An overview of the language

2.2.1 Basics

SIGNAL [37, 14] is a dataflow relational language that handles unbounded series of typed values \((x_t)_{t \in \mathbb{N}}\), called signals, implicitly indexed by discrete time, and denoted as \( x \). At any logical instant \( t \in \mathbb{N} \), a signal may be present or absent: when present, it holds some value; when absent, it holds no value and is denoted by \( \bot \) in the semantic notation. There is a particular type of signal called event. A signal of this type always holds the value \( true \) when it is present. The set of instants at which a signal \( x \) is present is referred to as its clock, noted \( \hat{x} \). A process is a system of equations over signals that specifies relations between values and clocks of the signals. A program is a process. The whole SIGNAL language relies on the six primitive constructs, presented below, which define elementary processes. Unlike the LUSTRE and LUCID SYNCHRONE languages in which a program expresses a functional dataflow specification, in SIGNAL, a program implicitly expresses constraints on the involved signals, that must be satisfied by their
values on the one hand and by their clocks on the other hand (see Table 1).

- **Instantaneous relations:** \( y := R(x_1, \ldots, x_n) \) where \( y, x_1, \ldots, x_n \) are signals and \( R \) is a point-wise n-ary relation/function extended canonically to signals. This construct imposes \( y, x_1, \ldots, x_n \) to be simultaneously present, and ii) to hold values satisfying the equation \( y := R(x_1, \ldots, x_n) \) whenever they occur. Its semantics is the following set of behaviors:

\[
\{ \begin{array}{l}
\forall t \in C, b(y)(t) = b(x_1)(t) \ldots = b(x_n)(t) \cap C \subseteq \emptyset \\
\forall t \in C, (b(x_1)(t), \ldots, b(x_n)(t)) \}
\]

where \( \emptyset \) is a point-wise interpretation of the relation \( R \).

- **Delay:** \( y := x \circ 1 \text{ init } c \) where \( y, x \) are signals and \( c \) is an initialization constant. This construct imposes \( i \) \( x \) and \( y \) to be simultaneously present while ii) \( y \) must hold the value carried by \( x \) on its previous occurrence. Its semantics is the following set of behaviors:

\[
\{ \begin{array}{l}
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset \\
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset 
\}
\]

- **Undersampling:** \( y := x \text{ when } b \) where \( y, x \) are signals and \( c \) is of Boolean type. This construct imposes \( i \) \( y \) to be present only when \( x \) is present and \( c \) holds the value \( \text{true} \), while ii) \( y \) must hold the value carried by \( x \) at those logical instants. Its semantics is the following set of behaviors:

\[
\{ \begin{array}{l}
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset \\
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset 
\}
\]

- **Deterministic merging:** \( z := x \text{ default } y \) where \( z, x, y \) are signals. This construct imposes \( i \) \( z \) to be present when either \( x \) or \( y \) are present while ii) \( z \) must always hold the value of \( x \) uppermost, otherwise it takes the value of \( y \). Its semantics is the following set of behaviors:

\[
\{ \begin{array}{l}
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset \\
\forall t \in C, b(y)(t) = b(x)(t) \cap C \subseteq \emptyset 
\}
\]

- **Composition of processes:** union of the equations defined in processes, leading to the conjunction of the constraints associated with these processes. For a given set of observation points \( T \), the composition on \( T \) of processes \( p_1 \) and \( p_2 \), noted \( p_1 \uplus p_2 \), is a process \( p = \{ (b, \{ \hat{b}_{\text{vars}(p_1)} \subseteq p_1, \hat{b}_{\text{vars}(p_2)} \subseteq p_2 \}) \} \) and \( \text{vars}(p) = \text{vars}(p_1) \cup \text{vars}(p_2) \).

Restriction (or Hiding): local declarations in a process. The restriction, noted \( p \text{ where } x \text{ (or } p/x \text{ for short)}, \) of a process \( p \) defined on \( X \subseteq T \) is a process defined on \( X \setminus \{x\} \) is defined as the following set of behaviors:

\[
\{ (b, \{ \hat{b}_{\text{vars}(p)} \subseteq p, \hat{b}_{\text{vars}(p_2)} \subseteq p_2 \}) \} \]

These constructs are expressive enough to derive new constructs of the language for comfort and structuring.

2.2.2 Clock calculus, endochrony and endo-isochrony

In SIGNAL, clocks are fundamentally the main means to express control-related properties, i.e. synchronizations between signals. They are ordered according to the following relation: a clock \( \kappa_1 \) is said to be greater than a clock \( \kappa_2 \), which is denoted by \( \kappa_1 \geq \kappa_2 \), if \( \kappa_2 \) is included in \( \kappa_1 \) in terms of sets of instants. The set of clocks associated with this relation is a lattice.

**Clock calculus.** The purpose of the clock calculus is to determine the existence of a greatest clock, called master clock, from which all clocks of a program can be extracted, and on the other hand, to verify the consistency of synchronization relations between signals. Nonetheless, in some programs, such a unique master clock may not exist.

**TABLE 1** Clock constraints in primitive constructs.

<table>
<thead>
<tr>
<th>Constructs</th>
<th>Clock constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y := f(x_1, \ldots, x_n) )</td>
<td>( y = x_1 = \ldots = x_n )</td>
</tr>
<tr>
<td>( y := x \circ 1 \text{ init } c )</td>
<td>( y = \hat{c} )</td>
</tr>
<tr>
<td>( y := x \text{ when } b )</td>
<td>( \hat{b} \cup { \hat{b} } = { \hat{b} } \cup \emptyset = \emptyset )</td>
</tr>
<tr>
<td>( z := x \text{ default } y )</td>
<td>( \hat{z} = \hat{y} \cup { \hat{y} } )</td>
</tr>
</tbody>
</table>

In this case, there are several local master clocks (see Section 3.1.2). **TABLE 1** gives the clock properties associated with primitive constructs of SIGNAL (here, \( \hat{b} \) resp. \( \hat{b} \) denotes the set of instant when the Boolean expression \( b \) is present and \( \text{true} \) resp. \( \text{false} \)). Such properties are automatically inferred by the compiler from any program to be analyzed. A specific option of the compiler enables to generate a file containing these properties under the form of clock relations defined in a SIGNAL program. Clock hierarchies can therefore be extracted from such a program.

**SIGNAL** allows one to explicitly manipulate clocks through some derived constructs such as the following ones (the equivalent expressions are given with primitive operators):

- **Sampling:** \( y := x \text{ when } b \) where \( b \equiv b \) when \( b \). It expresses the fact that \( b \equiv b \) occurs whenever \( b \equiv b \) is \( \text{true} \).
- **Clock union:** \( y := x_1 \cup x_2 \) where \( y := x_1 \text{ default } x_2 \). It denotes the set of instants at which at least one signal \( x_1 \) occurs. **Clock intersection** and **difference** are also defined.
- **Synchronizer:** \( x_1 \leq x_2 \) where \( x_1 := x_2 \) \( \cup \{ x : x_1 = x_2 \} \) / \( \{ x \} \). It means that \( x_1 \) and \( x_2 \) have the same clock or are synchronous.

**Endochrony and endo-isochrony:** key properties for program distribution.

The safe distribution of SIGNAL programs is possible thanks to two fundamental notions, **endochrony** and **endo-isochrony**, which have been formally defined in [37, 9].

In this paper, we only recall them informally and put them to work on our case study introduced in Section 1.3.

An endochronous program \( p \) is a program that can be executed in an environment, which only provides \( p \) with the values of its inputs, without any information about their status (present or absent). \( p \) is able to reconstruct a unique synchronous behavior from any external (asynchronous) input flow of values. It is insensitive to external propagation delays. From the viewpoint of clocks, such a program holds a master clock \( \kappa \) that plays the role of its activation clock. All the other clocks of \( p \) are necessarily defined recursively from \( \kappa \). They are smaller than \( \kappa \) following the partial order \( \leq \). The global clock hierarchy in \( p \) therefore forms a tree where the root node represents the master clock of \( p \) (see Section 3.1.1). Given any SIGNAL program \( p \), the compiler is able to check whether or not \( p \) is endochronous. An endochronous program is **deterministic:** for the same functional input values it yields the same functional output values whatever the external propagation delays are. The elementary processes defined by the instantaneous relations and delay primitive constructs are examples of very simple endochronous programs. All signals involved in these processes have the same clock, which is also the master clock. On the contrary, the elementary processes defined by the undersampling and merging primitive constructs are not endochronous programs since there is no master clock from which the clocks of involved signals can be inferred in
endochrony of the different sub-parts of the composition.

2.2.3 A rich tool-set for validation

The Signal design environment, Polychrony [14], offers a compiler and a model-checker that support the trustworthy

dependence on their common variables) to be endochronous as

Thus, p1 and p2 can be safely deployed on GALS architectures.

2 can be somehow seen as an initialization period value of
cnt. The Signal start_confirm denotes the logical instants
at which a confirmation begins, described by equations at lines
8 and 9. It occurs when the counter value reaches some amount
of time denoted by delay within each cycle.

Note that the sub-process Alarm_Confirm, defined at line
15 and invoked at line 10, to produce the output alarm_out,
uses a derived construct of SIGNAL, referred to as array
of processes, that enables to describe instantaneous iterations [14].

The body of the process is composed of five equations,

from line 5 to line 11. The local signals cnt, zcnt and
start_confirm are used to specify when the process con-
firms alarms in equations at lines 5 to 8. The signal cnt plays
the role of a counter of logical instants. It is set to k−1 whenever
its previous value zcnt becomes zero; otherwise its previous
value is decremented by one. The static parameter k at line
2 can be somehow seen as an initialization period value of
cnt. The Signal start_confirm denotes the logical instants
at which a confirmation begins, described by equations at lines
8 and 9. It occurs when the counter value reaches some amount
of time denoted by delay within each cycle.

Note that the sub-process Alarm_Confirm, defined at line
15 and invoked at line 10, to produce the output alarm_out,
uses a derived construct of SIGNAL, referred to as array
of processes, that enables to describe instantaneous iterations [14].

From the above equations and according to Table 1, we can
deduces the following system of clock constraints associated
with the process Alarm_Manager (for a signal s, we denote by clk_s its associated clock):

$$
\begin{align*}
\text{clk}_{\text{cnt}} & = [\text{cnt} = 0] \cup \text{clk}_{\text{cnt}} \quad \text{(lines 5 & 6)} \\
\text{clk}_{\text{zcnt}} & = \text{clk}_{\text{cnt}} \quad \text{(line 7)} \\
\text{clk}_{\text{start\_confirm}} & = [\text{cnt} = \text{delay}] \quad \text{(line 8)} \\
\text{clk}_{\text{alarm\_in}} & = \text{clk}_{\text{start\_confirm}} \quad \text{(line 9)} \\
\text{clk}_{\text{alarm\_out}} & = \text{clk}_{\text{alarm\_in}} \quad \text{(lines 10 & 19)}
\end{align*}
$$

After reductions of the above system 1, we obtain the
following clock hierarchy:

$$
\text{clk}_{\text{cnt}} \equiv \text{clk}_{\text{zcnt}}
$$

and

$$
[zcnt \neq \text{delay}] \equiv [\text{cnt} = \text{delay}] \equiv \text{clk}_{\text{start\_confirm}} \equiv \text{clk}_{\text{alarm\_in}} \equiv \text{clk}_{\text{alarm\_out}}
$$

We can see that the master clock is the one of cnt, which
is the same for zcnt. The other clocks of the process are
defined following a partitioning of the values of zcnt w.r.t. the
value of delay. As a result, the process Alarm_Manager is
endochronous (hence deterministic). In a similar way, we prove that the process Alarm_Notifier is also endochronous. More generally, the SIGNAL compiler allows one to automatically check the endochrony property of a process based on this technique and to generate an associated simulation code.

### 3.1.2 Global model of the FWS system

```
1  process FW_System =
2  { integer k1,k2,d1,d2 }
3  { ? [n] alarm_type alarm_in;
4    ! [n] alarm_type alarm_out; }
5  { integer cnt, zcnt;
6    default (zcnt = 0)
7    zcnt := cnt $ init 0
8    start_confirm := when(zcnt = delay)
9    alarm_in := start_confirm
10   alarm_out := Alarm_Confirm(alarm_in)
11 }
12 where
13   integer cnt, zcnt;
14   event start_confirm;
15   process Alarm_Confirm =
16     { ? [n] alarm_type alarm_in;
17       ! [n] alarm_type alarm_out;
18     }
19   { array i to n-1 of
20     out[i].conf := in[i].pres
21     end; }
22 end;
```

Fig. 6. Excerpt of the SIGNAL code of FWS.

The process depicted by Fig. 6 represents the FWS model. It takes as input a collection of alarms represented by alarm (line 3), and produces as outputs the signals denoted by s_0, ..., s_n-1. These outputs are generated by Alarm_Notifier.

The equations from lines 5 to 7 define the interaction between the concurrent processes in the FWS system: the input of process Alarm_Notifier are the output of process Alarm_Manager. This transfer of data is realized via the local signal tmp. It means that the two processes must agree on the set of specific instants at which this transfer is possible.

Let us analyze the clock properties induced by the definition of process FW_System. We obtain the following two-rooted clock hierarchy:

```
dclk_cnt1 ≡ dclk_zcnt1
```

We clearly distinguish the clock trees associated with Alarm_Manager and Alarm_Notifier. There is no master clock because the clocks of the sub-processes (clk_cnt1 for Alarm_Manager and clk_cnt2 for Alarm_Notifier) are included in none of each other. However, they share common instants at which the clock transfer is done between the manager and the notifier processes. Such instants are defined only when both zcnt in Alarm_Manager and zcnt in Alarm_Notifier are respectively equal to d1 and d2. Such a requirement appears under the form of a clock constraint generated by the SIGNAL compiler. The corresponding code which is automatically produced (~ force option of the compiler) from such a model contains exception messages that are thrown whenever the clock constraints are not satisfied during code execution.

In such a situation, it means that either the environment of the two concurrent processes is able to guarantee that they can synchronize at the required specific logical instants, and the model will execute as expected; or it is not possible, and the output of the whole system model is undefined whenever the clocks of the two processes fail to agree. In SIGNAL, the special construct called assert offers the way to define assumptions in programs [14]. It could be used here to meet the requirement on the concurrent execution of Alarm_Manager and Alarm_Notifier processes.

On the other hand, one can think of another specification of the FW_System model in which the confirmed alarm values tmp are transferred via a "clock-less" memory, which is made available whenever a process needs to make an access to this memory. This is obtained very easily by introducing a new equation in the body of FW_System, which defines a local signal tmp_mem as follows: tmp_mem := (var tmp). Then, instead of putting tmp as input for Alarm_Notifier at line 7 in Fig.6 we consider tmp_mem. The var operator allows one to memorize the value of tmp in tmp_mem [14]. The clock of tmp_mem is the one of the context in which it is used, i.e., the memorized value is available whenever required.

In this second version of the FW_System model, there is...
no longer clock constraints on the specific instances at which the concurrent processes must exchange information. Their associated master clocks are completely independent, with a possible empty set of common instants. In this case, the compiler generates no clock constraints and automatically produces a code without any exception message.

A very interesting feature of the FW_System model is that it satisfies the endo-isochrony property: first, Alarm_Manager and Alarm_Notifier models are both endo-isochronous; second, their communicating part which consists of the single signal tmp (see Fig. 5) is trivially endo-isochronous. Indeed, a single signal yields a clock hierarchy with only one clock node, which consequently forms a clock tree. The clock hierarchy associated with FW_System typically illustrates the endo-isochrony of a pair of processes. In particular, the synchronous communication between Alarm_Manager and Alarm_Notifier via tmp can be equivalently replaced by an asynchronous communication while still preserving the semantics of the FW_System model.

At this stage, some preliminary verification can be done on the FW_System model in order to make sure that it conforms to its expected specification requirements. Typically, one may check the absence of null clocks in the defined model using the compiler. The presence of such clocks denote a possible absence of reactions during the execution of the program. This is often undesirable for reactive systems.

### 3.2 Architecture model and manual distribution

The hardware architecture on which the previous application functionality model will be mapped is now defined. For the sake of simplicity, only processors are represented. Concretely, each processor is graphically modeled as an empty box.

Afterwards, one can proceed to the mapping of the application functionality on the hardware architecture in the POLYCHRONY graphical user interface. A SIGNAL code can be duplicated in different processor boxes; so, code replication is allowed. From this mapping, a new description of the system denoted by $p' = p_1 | ... | p_m$ is obtained, which reflects the target hardware architecture ($m$ denotes the number of processors). Fig. 7 illustrates the distribution of the FWS model on a two-processors architecture, i.e. $m = 2$.

![Fig. 7. Graphical distributed model of the FWS system.](image)

We can notice that the sub-processes $p'_j$ may not be necessarily endo-isochronous: after the mapping, $p'_j$ can be obtained from the composition of different sub-processes $p_i$ characterized by different clock trees without a common root clock.

### 3.3 Automatic transformations

Let us consider the model $p'$ resulting from the mapping of the application functionality on the processors. The main purpose is to transform the process $p'$ so that: i) $p'$ fully preserves the functional semantics of $p$ and ii) the sub-processes $p'_j$ become endo-isochronous and hence becomes endo-isochronous.

So, each process $p'_j$ is compiled modularly in the context of $p$: from the clock hierarchy of $p$, the clock hierarchy of $p'_j$ is transformed in such a way that it becomes a clock tree, i.e. $p'_j$ becomes endo-isochronous. For that, Boolean interface signals are used (each root node in a clock hierarchy of a non endo-isochronous process is associated with a Boolean signal, and all these signals are made synchronous). On the other hand, some transformations are performed in order to render the communication part endochronous between the endochronous processes $p'_j$. For that, additional communicating signals are inserted between processes, which enable each process $p'_j$ to receive all information it requires to execute. The result of these transformations is carried out automatically on the graphical representation of $p'$ within the POLYCHRONY editor. In the obtained transformed process, denoted by $p''$, the information exchanges between processes are described by instantaneous communications. Here again, possible verification techniques can be applied to $p''$ in order to ensure that it still preserves the functional properties of $p$.

For our FWS system model, all the above transformations are not necessarily required since FW_System is already endo-isochronous as shown before. The only effect of these transformations is the automatic insertion of the communicating ports on processor boxes and the connections between these ports.

### 4 Deployment on specific platforms

In order to refine the descriptions obtained from the previous steps of the methodology, a set of predefined components is proposed, which is usable to instantiate various aspects of the system: asynchronous communication mechanisms such as the FIFO buffer modeled in this paper, execution supports (e.g., tasks and processes), system-level primitives for task management as well as for synchronization and inter-task communication services (e.g. the APEX service models). These components have been modeled in SIGNAL and are grouped in a library available in the POLYCHRONY platform.

#### 4.1 Design of a library of components

We consider the FIFO queue presented in [21], to show how asynchronous communication mechanisms can be defined in SIGNAL, and proved to be correct. This queue is well adapted as mechanism to model a message buffer on which send/receive requests can be achieved by endo-isochronous processes. In addition, in the architecture level, it will be easily extended to model a bus. Compared to [21], here we give a more complete presentation of the design and analysis. A first model of FIFO queue is presented (basic_FIFO); then, a more constrained version (safe_FIFO) is defined from the first one and its key properties are checked.

##### 4.1.1 A basic FIFO queue

We call basic FIFO, a message queue that works as follows.

On a write request, the incoming message is inserted in the queue regardless of its size limit. When the queue was previously full, the oldest message is lost. The other messages are shifted forward, and the incoming one is inserted in the queue.

On a read request, there is an outgoing message whatever the queue status is. When it was previously empty, two situations are distinguished: if there is not yet any written message,
an arbitrary message called \textit{default message} is returned; else the outgoing message is the message that has been read last. Furthermore, for simplicity we suppose that simultaneous write/read requests on the queue never occur.

The \texttt{ SIGNAL} model corresponding to the basic FIFO is given in \textbf{Fig. 8}. It is defined as a \texttt{ SIGNAL} process, represented by the keyword \texttt{process} (line 1). The static parameters \texttt{msg_type}, \texttt{size} and \texttt{def_msg}, declared at line 2, respectively denote the type of messages, the size limit of the queue, and the default message value. The input signals \texttt{msg_in} and \texttt{access_clock} (line 3), are respectively the incoming message (its presence denotes a write request), and the queue access clock (i.e. instants of read/write requests). The output signals (line 4), are \texttt{msg_out}, \texttt{nbmsg}, \texttt{OK_write} and \texttt{OK_read}. They respectively represent the outgoing message, the current number of messages in the queue, and conditions for writing and reading.

Now, we can take a look at the meaning of the statements in the process body. Let us begin with the equation at line 10; it defines the local signal \texttt{prev_nbmsg}, which denotes the previous number of messages in the queue. This signal is used in equations at lines 8 and 9 to define respectively when the queue can be “safely” written (the size limit is not reached), and read (there is at least one message received). This is the meaning of the signals \texttt{OK_write} and \texttt{OK_read}.

The statement at lines 5 and 6 expresses how the current number of messages is calculated. The previous value of this number is incremented by one on a write request when the queue was not full. It is decremented by one on a read request when the queue was not empty. Otherwise, it remains unchanged. The equation at line 14 states that the value of \texttt{nbmsg} is defined whenever there is a request on the queue.

The equation at line 10 defines the message queue. The signal queue is an array of dimension \texttt{size} that contains the size latest values of \texttt{msg_in} (expressed by the window operator). The cell operator makes the signal queue available when \texttt{access_clock} is present. The semantics of these derived constructs is as follows:

- **Sliding windows.** In \texttt{$y := x$ window k init y init}, the signal \texttt{y} is an array of size $k \geq 1$ whose elements have the same type as signal \texttt{x}; and \texttt{y init} is an array of dimension $k' \geq k - 1$, containing initialization values. This equation defines a sliding windows on \texttt{x}, of constant size \texttt{k} such that:

\[
\forall t \geq 0 \Rightarrow (| y := x$ window k init c \rangle \land (1 \leq t + i < k) \Rightarrow (y[i] = y [i] = y init [t - k + i + 1])
\]

- **Memory.** The expression \texttt{y := x cell b init c} enables to \texttt{memorize} the values of the signal \texttt{x}. It is defined as:

\[
| y := x$ cell b init c \rangle \\
| y := x$ cell b init c \rangle
\]

The signal \texttt{y} takes the value of \texttt{x} when \texttt{x} is present. Otherwise, when \texttt{x} is absent and the signal \texttt{b} is present and holds the value \texttt{true}, \texttt{y} memorizes the latest value of \texttt{x}. When \texttt{b} is present and \texttt{true} before the first occurrence of \texttt{x}, \texttt{y} is initialized with the constant \texttt{c}. The clock of \texttt{y} is the union of \texttt{b} and \texttt{c}.

Finally, the statement at lines 11 and 12 means that on a read request (i.e. at the clock \texttt{msg_out}), the outgoing message is either the previous one if the FIFO is empty (defined at line 13), or the oldest message in the queue.

### 4.1.2 A safe FIFO queue

In the model depicted in \textbf{Fig. 9} the interface is slightly different from that of basic FIFO. The new input signal \texttt{get mess} denotes a read request. The signal \texttt{nbmsg} is local.

The statement at line 9 defines the access clock as the union of instants at which read/write requests occur. Equations at lines 10 and 11 specify the safe access to the queue in basic FIFO. The process call at lines 12, 13 and 14 has the local signal \texttt{new_msg_in} as input. This signal is defined only when basic FIFO was not full (line 10). The equation at line 11 expresses that on a read request, a message is enqued only when basic FIFO was not empty. In the trace in \textbf{Fig. 9} the same parameters as for basic FIFO are considered.

Through the above modeling approach, we observe that modularity and reusability are key features of the \texttt{ SIGNAL} programming. They favor component-based design.

\textbf{Fig. 10} shows the FWS model in which the communication between \texttt{Alarm_Manager} and \texttt{Alarm_Notifier} is achieved via the \texttt{safe_FIFO} model. Note the modification in the \texttt{Alarm_Notifier} at line 27 where the new output signal \texttt{get_msg} denotes read requests on \texttt{safe_FIFO}.

```
process basic_FIFO =
| type msg_type; integer size; msg_type def_msg; |
| msg_type msg_in; event access_clock; |
| msg_type msg_out; integer nbmsg; boolean OK_write, OK_read; |
| nbmsg := ((prev_nbmsg + 1) when ("msg_in") when OK_write) |
| default ((prev_nbmsg - 1) when ("msg_out") when OK_read) default prev_nbmsg |
| prev_nbmsg := nbmsg$1 init 0 |
| OK_write := prev_nbmsg ≥ size |
| OK_read := prev_nbmsg ≤ 0 |
| msg_out := msg_out when (not OK_read) when (msg_out) |
| default queue[size - prev_nbmsg] when (msg_out) |
| prev_msg_out := msg_out $ 1 init def_msg |
| nbmsg := access_clock |
| ... |
where |
integer prev_nbmsg; [size]msg_type queue; msg_type prev_msg_out;
end;
msg_in : ⊥ 4 6 ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ..., |
access_clock : t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t ... |
msg_out : -1 ⊥ ⊥ ⊥ ⊥ 4 6 6 ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ 7 8 ... |
nbmsg : 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0 0 1 2 1 0
Fig. 9. Model of the safe FIFO with an execution trace.

Fig. 10. Excerpt of the FWS model including safe_FIFO.

4.2 Ensuring the reliability of components

4.2.1 Safety and fairness

To check the properties of the safe_FIFO process, we consider an abstraction based on its state variables, i.e., signals defined by delay or memory operators in the process. These signals feature the dynamics of the system defined by the process. Here, the state variables are nbmsg, queue and prev_msg_out (defined in basic_FIFO). They entirely reflect the dynamics of the equation system associated with safe_FIFO. Let us represent the state of this system by \( \sigma \). When the queue is full (resp. empty), the value of \( \sigma \) is noted full (resp. empty); otherwise it is equal to none. Let \([\text{safe_FIFO}]\) denote the set of behaviors associated with safe_FIFO and \( b \) a behavior that belongs to this set, we address the following important properties of the safe_FIFO in order to guarantee its robustness:

Property 1 (safe read requests): The safe_FIFO queue never delivers an output message on a read request whenever it is empty (and its state remains unchanged): \( \forall b \in [\text{safe_fifo}], T = \text{tags}(b(\text{get_msg})) \)

\[
t \in T \land t \neq \min(T) \land b(\sigma)(\text{prey}(t)) = \text{empty}
\Rightarrow b(\sigma)(t) = b(\sigma)(\text{prey}(t)) = \text{empty} \land t \notin \text{tags}(b(\text{msg_out}))
\] (2)

Property 2 (safe write requests): The safe_FIFO queue never accepts an input message on a write request whenever it is full (and its state remains unchanged): \( \forall b \in [\text{safe_fifo}], T = \text{tags}(b(\text{msg_in})) \)

\[
t \in T \land t \neq \min(T) \land b(\sigma)(\text{prey}(t)) = \text{full}
\Rightarrow b(\sigma)(t) = b(\sigma)(\text{prey}(t)) = \text{full} \land t \notin \text{tags}(b(\text{msg_in}))
\] (3)

We recall that the signal \( \text{new_msg_in} \), which is defined in the safe_FIFO process, at line 10, denotes incoming messages that are effectively enqueued.

Property 3 (fair read requests): A message can always be read from the safe_FIFO queue, whenever it is not empty: \( \forall b \in [\text{safe_fifo}], T = \text{tags}(b(\text{get_msg})) \)

\[
t \in T \land t \neq \min(T) \land b(\sigma)(\text{prey}(t)) = \text{empty}
\Rightarrow t \in \text{tags}(b(\text{msg_out}))
\] (4)

Property 4 (fair write requests): A message can always be written in the safe_FIFO queue, whenever it is not full: \( \forall b \in [\text{safe_fifo}], T = \text{tags}(b(\text{msg_in})) \)

\[
t \in T \land t \neq \min(T) \land b(\sigma)(\text{prey}(t)) = \text{full}
\Rightarrow t \in \text{tags}(b(\text{new_msg_in}))
\] (5)

The values of \( \sigma \), the state of the system corresponding to safe_FIFO, can be defined by considering the signals \text{OK_write} and \text{OK_read}. The definition of these signals is itself based on the current number of enqueued messages, represented by \( \text{nbmsg} \). Thus, \( \sigma \) can be characterized, using \( \text{nbmsg} \), as follows: \( \forall b \in [\text{safe_FIFO}] \forall t \in \text{tags}(b), \)

\[
b(\sigma)(t) = \text{full} \Leftrightarrow b(\text{nb_msg})(t) = \text{size}
\]

\[
b(\sigma)(t) = \text{empty} \Leftrightarrow b(\text{nb_msg})(t) = 0
\]

\[
b(\sigma)(t) = \text{none} \Leftrightarrow 0 < b(\text{nb_msg})(t) < \text{size}
\] (6)

Since SIGALI does not address numerical properties, we must also find a Boolean encoding (or abstraction) for \( \text{nbmsg} \).
4.2.2 Abstraction and verification by model-checking

As shown in Fig. 11, a n-FIFO queue can be represented by an automaton with \((n + 1)\) states, where a state denotes the current number of messages in the queue. For the sake of simplicity, we consider a 2-FIFO queue since all possible relevant configurations can be addressed. So, the result remains valid for any bounded n-FIFO queue where \(n > 2\).

The automaton illustrated in Fig. 11 represents a behavior abstraction of a n-FIFO queue by a 2-FIFO queue. It relies on the behavioral isomorphism between both FIFO queues, which can be trivially observed. A state \(s_k\) (represented by a circle) denotes that the queue contains \(k\) messages, i.e. \(\forall k \in \{0, 1, 2\}\):

\[
(n_{msg} = k \Rightarrow s_k = true) \land (n_{msg} \neq k \Rightarrow s_k = false)
\]

The state \(s_0\) represents the initial state. Labels \(msg_{in}\) and \(msg_{out}\) are respectively write and read requests. Two special states (represented by rectangles) have also been added. They characterize “illegal” accesses to the queue: ERR_empty is reached on an attempt to read an empty queue, and ERR_full is reached when overwriting a full queue. They are also encoded by Boolean variables. Automata are very easy to specify in SIGNAL. For instance the state \(s_0\) and its associated transitions are defined as follows:

\[
\begin{align*}
& |s_0| := \text{true when prev_s1 when } (msg_{out}) \\
& \text{default } (\text{false when prev_s0 when }) \\
& (msg_{in} \Rightarrow msg_{out}) \text{ default } prev_s0 \\
& |prev_s0 := s_0| = \text{init true } |
\end{align*}
\]

All the other states are specified in a similar way. It follows the definitions of signals OK_write and OK_read below:

\[
\begin{align*}
& |OK_write := \text{false when (prev_err_full or prev_s2)}\text{ default } true | \\
& |OK_read := \text{false when (prev_empty or prev_s0)}\text{ default true } |
\end{align*}
\]

The first equation means that a write request is not authorized when there are already two messages in the 2-FIFO queue (\(prev_s2\) is true), or when the queue has been overloaded previously (\(prev_{err\_full}\) is true); otherwise it can be accepted. In a same way, the other statement specifies when a read request is legal.

The signals \(s_0, s_1, s_2, ERR_{empty}, ERR_{full}, OK_{write}\) and \(OK_{read}\) are synchronized with access_clock.

The reachability of “error” states (i.e. safety properties) is concretely checked with SIGNAL by considering the script given in \[21\].

To verify properties \(3\) and \(4\) we consider observers, represented by Boolean state variables. We have to show that these variables are always true. Let \(obs_4\) and \(obs_3\) denote respectively the observers for Properties \(3\) and \(4\).

- Property \(4\) is described as follows:
  - On a write request (denoted by the presence of \(msg_{in}\)), when the queue is either in \(s_0\) or \(s_1\); the signal \(obs_4\) carries the value true if the message is actually written into the queue (i.e. \(new_{msg}_{in}\) is present), else \(obs_4\) is false.
  - Otherwise, \(obs_4\) keeps its previous value.

The corresponding SIGNAL code is:

\[
\begin{align*}
& |\text{actual}_{w} := \text{true} \text{ when } (\text{new}_{msg}_{in}) \text{ default } false | \\
& |obs_{4} := \text{actual}_{w} \text{ when } (s_{0} \text{ or } s_{1}) | \\
& |\text{msg}_{in} := \text{false when } (\text{prev}_{err}_{empty} \text{ or } s_{2}) \text{ default } z_{obs_{4}} | \\
& |z_{obs_{4}} := obs_{4} \text{ when } s_{0} | \\
\end{align*}
\]

The Boolean \(\text{actual}_{w}\) denotes the fact that a message is actually inserted in the queue.

- In a similar way, Property \(3\) is encoded by the following SIGNAL code:

\[
\begin{align*}
& |\text{actual}_{r} := \text{true} \text{ when } (\text{msg}_{out}) \text{ default } false | \\
& |\text{obs}_{3} := \text{actual}_{r} \text{ when } (s_{1} \text{ or } s_{2}) | \\
& |\text{get}_{msg} := \text{false when } (\text{prev}_{err}_{empty} \text{ or } s_{2}) | \\
& |z_{obs_{3}} := obs_{3} \text{ when } s_{0} | \\
\end{align*}
\]

Here also, all the new variables have the same clock as the signal access_clock. Then, the fairness properties can be checked as shown in \[21\].

4.2.3 Queueing order preservation

We have to prove that the safe_FIFO queue actually preserves the arrival order of enqueued messages. The idea consists in considering the flow of input messages in the call to basic_FIFO at line 13 in FIG. 9. Such a flow corresponds to that of signal new_msg_in. Indeed, only the values of this signal are taken into account and read afterwards. We therefore need to verify the following property:

Property 5 (order preservation):

- \((E_1)\); the order of values present in the input flow is preserved by the definition of the message queue: \(\forall b \in [safe_FIFO] \forall v, v' \in V \exists t, t' \text{ with } v = b(new_{msg}_{in})(t), v' = b(new_{msg}_{in})(t') \text{ and for } i, i' \text{ the respective indices of } v, v' \text{ in queue}\text{ the following holds: } t < t' \Rightarrow i \leq i'.\]

- \((E_2)\); the order of values in the output flow is the same in the message queue: \(\forall b \in [safe_FIFO] \forall v, v' \in V \exists t, t' \text{ with } v = b(msg_{out})(t), v' = b(msg_{out})(t') \text{ and for } i, i' \text{ as above, the following holds: } i \leq i' \Rightarrow t \leq t'.\]

The order preservation property is specifically related to the properties of specific SIGNAL constructs used in the FIFO queue model. A proof of this property is in Appendix Section.

4.3 Use of safe_FIFO in architecture model

We have shown that the above F_W_System model can be safely described with asynchronous communication mechanisms between its concurrent processes Alarm_Manager and Alarm_Notifier. Now, let us imagine a deployment of this system on a given architecture as illustrated in FIG. 9.

In this architecture, there are two processors Processor 1 and Processor 2 that communicate via a bus called Bus. The main feature of such an architecture that is taken into account in our

2. Here, the enqueued values are assumed to be inserted from left to right in the array defined with the window operator.
models is clock constraint. Each of the three components holds its own activation clock, which is different from those of others. Let us denote by \( \kappa_1 \), \( \kappa_2 \) and \( \kappa_3 \) the clocks corresponding respectively to Processor 1, Bus and Processor 2.

The bus model considered here is obtained from the previous safe_FIFO model on top of which a master clock is defined (see Fig. 12). This clock \( \kappa_1 \) is represented by \( \text{cnt} \) in the Bus process. It enables to define the instants at which incoming and outgoing transferred data are respectively accepted and delivered by the bus. This is specified through the signal \( \text{enable_in_out} \) at line 11. The static parameter \( \text{rw_per} \) denotes the period at which a read/write request is accepted by the bus. Its value belongs to the interval \( 0..k-1 \) (note that \( k \) is also a static parameter in Bus).

\begin{verbatim}
process Bus =
{ type msg_type; integer size;
  msg_type def_msg; integer k, rw_per;
  event get_msg;
  ! integer msg_out;
  boolean OK_write, OK_read;
  (\text{cnt}:= ((k-1) when (\text{zcnt} = 0))
  default (\text{zcnt} - 1)
  \text{zcnt}:= \text{cnt} \& init 0
  when \text{enable_in_out}:= when(\text{zcnt} = \text{rw_per})
  \text{msg_in} ^+ \text{get_msg} ^=
  \text{enable_in_out}:
  \text{msg_out}, \text{OK_write}, \text{OK_read}:=
  \text{safe_FIFO}(\text{msg_type}, \text{size}, \text{def_msg})
  (\text{msg_in}, \text{get_msg})
  |
  \}
where
use safe_FIFO; integer cnt, zcnt;
end;
\end{verbatim}

Fig. 12. A bus model in SIGNAL.

On the other hand, we assume that the Alarm_Manager and Alarm_Notifier processes run at the frequency of their allocated processors: the master clock of Alarm_Manager is the same as \( \kappa_1 \) while the master clock of Alarm_Notifier is the same as \( \kappa_2 \). So, processor constraints are implicitly taken into account by the Alarm_Manager and Alarm_Notifier models, which consequently remain unchanged.

Finally, the overall model of the deployed system is close to the one shown in Fig. 10 where safe_FIFO is replaced by Bus. Its clock analysis by the compiler exhibits the clock constraints that should be guaranteed by the program environment in order to perform functional simulation.

5 Analysis and Automatic Code Generation

5.1 Analysis of non functional properties

There are various ways to analyze non functional properties of SIGNAL models after their deployment on a given platform. For instance, performance evaluation is possible by using a quantitative temporal interpretation of SIGNAL programs [36], [33]. On the other hand, beyond the strict clock synchronization problems that are usually solved by almost all compilers of synchronous languages, more intricate synchronization issues can arise in the design of distributed systems if one would like to be able to widely explore architecture allocation possibilities. This requires sophisticated tools to address such issues.

5.1.1 Clock synchronizability issues

Affine clocks have been introduced in the SIGNAL clock calculus for this purpose [43], and more precisely in order to deal with synchronizability issues in a more “relaxed” way than usually in synchronous languages. Formally, an affine transformation of parameters \((n, \phi, d)\) applied to a clock \( \kappa_1 \) produces a clock \( \kappa_2 \) by inserting \((n-1)\) instants between any two successive instants of \( \kappa_1 \), and then counting on this fictional set of instants each \( d^{th} \) instant, starting with the \( \phi^{th} \). Clocks \( \kappa_1, \kappa_2 \) are said to be in \((n, \phi, d)\)-affine relation, noted as \( \kappa_1 \overset{(n, \phi, d)}{\rightarrow} \kappa_2 \). The following trace depicts \( \kappa_1 \overset{(3,1,4)}{\rightarrow} \kappa_2 \), where "\( i \)" denotes the presence of a clock \( \kappa_i \):

\[
\begin{array}{cccccccccccc}
\text{instants} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \ldots \\
\kappa_1: & t & \perp & \perp & t & \perp & \perp & t & \perp & \perp & t & \ldots \\
\kappa_2: & t & \perp & \perp & t & \perp & \perp & t & \perp & \perp & t & \ldots \\
\end{array}
\]

In affine clock systems, two different clocks are said to be synchronizable if there is a dataflow preserving way to make them actually synchronous.

To show how such a notion can help to address non-trivial design decisions, let us consider the following allocation choices needed by a designer:

- to allow for a coherent communication protocol in this architecture, the following affine relations are assumed:
  \[
  \kappa_1 \overset{(1,\phi_1,d_1)}{\rightarrow} \kappa_2 \text{ and } \kappa_3 \overset{(1,\phi_2,d_2)}{\rightarrow} \kappa_4,
  \]
- to satisfy some production-consumption rate which enables the Alarm_Manager process to confirm a certain number of alarms before their notification by the Alarm_Notifier process, an affine relation is assumed between their respective clocks \( \text{clk_cnt1} \) and \( \text{clk_cnt2} \):
  \[
  \overset{(1,\phi_3,d_3)}{\rightarrow} \text{clk_cnt2}. \]

While the first constraint is related to the architecture, the second one is purely functional.

Now, let us assume that only the clock \( \text{clk_cnt1} \) of the Alarm_Manager process is identical to the clock of its associated processor, i.e. \( \kappa_1 \). So, the main question is which parameter values in the affine relations allow the designer to guarantee the synchronizability of the clocks \( \text{clk_cnt2} \) and \( \kappa_4 \) with respect to the above architectural and functional constraints.

The answer is given by solving the following system:

\[
\begin{align*}
\phi_1 + d_1\phi_2 &= \phi_3 \\
d_1d_2 &= d_3 
\end{align*}
\]
about proofs. So, BMC is clearly not sufficient alone but is a good complement to the other verification tools.

5.1.2 Applicability to the design of specific systems

Integrated Modular Avionics. Integrated Modular Avionics (IMA) \cite{3} are typical distributed systems that can greatly benefit from the synchronous multi-clock modeling. In particular, the SIGNAL clock synchronizability analysis offers an interesting basis to deal with the very challenging issue of finding suitable resource allocation strategies in IMA architectures \cite{2}.

In such architectures \cite{3}: i) time and memory resources are divided into partitions, which are associated with different functions at run-time; ii) partitions are composed of processes, representing the execution units that communicate within the same partition or with processes from other partitions. Partitions are scheduled according to a static cyclic policy while process scheduling is priority preemptive.

IMA-based implementation platforms assume the specific ARINC 629 Data Bus \cite{1} and the 659 Backplane Data Bus \cite{2}. Both buses adopt cyclic time-triggered message scheduling. In such a context, considering affine clocks can be significantly worth to designers to address the consistency of, on the one hand partition allocation to cyclic processors, and on the other hand cyclic processor schedules with ARINC 629 and 659 buses. This can be carried out on an IMA-based design of the FWS system, where processes are associated with partitions \cite{25} that execute on different processors communicating via ARINC 629 and 659 buses. The system \cite{7} enables to address partition allocation and communication constraints.

Further important implementation platforms for which the previous synchronizability analysis is very useful are those achieving communications with the Avionics Full Duplex Switched Ethernet (AFDX) technology \cite{A380} as in the Airbus A380. AFDX is very interesting because it is faster than its predecessors such as ARINC 629. It is based upon Ethernet and TCP/IP general principles. It is not time-triggered unlike the above two buses. Accordingly, with such a technology, dealing with the correct synchronization for the safe communication is more difficult. We believe that the polychronous model could be used to describe partial clock relations that capture the main synchronization points in an AFDX-based system. Then, the available analysis tools and techniques can be again applied to reason about the system behavior.

Large scale system-on-chip. The polychronous model is a good solution to deal with the shortcomings of GALS design frameworks in general, which adopt ad hoc methods where synchronous components are encapsulated with wrappers and communications are achieved by handshake. As illustrated in this paper, it allows one to uniformly model both aspects and formally validate the global system w.r.t. its correctness requirements. In addition, its vision of the synchrony-asynchrony link is suitable for large scale system-on-chip design \cite{22}.

5.2 Automatic code generation

After the non functional analysis of the deployed model of the FWS system, we can now consider an automatic modular code generation. Since, each “processor” is endochronous, the associated code can be generated independently from the other processors. The same holds for communications. This generation is necessarily achieved in the form of clusters \cite{15}. In SIGNAL, a cluster denotes a group of statements that depend on the same sub-set of input data. This type of code generation is particularly interesting in that it enables an efficient execution for each processor. Indeed, the operating system supported by a processor is solicited less often because of the reduced number of commutations. The code corresponding to clusters is generated in various C, C++ or Java. In the current experimental implementation of the code generation process in POLYCHRONY, the obtained code is mainly intended for functional simulation of distributed applications.

6 Discussions

All the design steps and analyses presented in our methodology are achieved within the POLYCHRONY environment \cite{14}, which provides the required tools, e.g. the SIGNAL compiler and the SIGALI model-checker. Along this presentation, we have stressed the advantages of our proposition for the reliable design of distributed embedded systems.

First, compared to mainstream implementation approaches that may rely for instance on C or Java programming, the synchronous programming offers well adapted specification concepts enabling a designer to safely describe systems. As an example, let us consider the endochrony property. Given a program \( p \) that satisfies such a property, \( p \) is able to read its required input data at the appropriate instants, based on its current state, from any asynchronous streams of input data. There are straightforward conditions in the compilation of SIGNAL programs, enabling to characterize this property of \( p \): when the resulting overall clock hierarchy of \( p \) yields a tree of clocks. This property is not necessarily easy to achieve in mainstream approaches. In such approaches, test mechanisms for checking the availability of the input data required by a program must be defined. Endochrony will be therefore obtained by associating a given function in C or Java with an additional piece of code conferring this function the ability to decide alone which input data must be read during its execution.

There are further benefits of synchronous programming over the use of synchronization mechanisms such as monitors in a language like Java. In synchronous programs, the fact that at any instant, an object, i.e. a signal, is written only once is guaranteed by construction. So, one does not need to consider some additional protection mechanism to enforce safe writes. As a result, the risk to get errors related to the usage of such delicate mechanisms is avoided. We can also mention the important role that play the formal tools and techniques provided by the synchronous technology for a trustworthy validation of designs.

From a methodological point of view, an interesting feature of our proposition is that it promotes correct-by-construction design: there is no need to (thoroughly) check the correctness of a system after design. This significantly reduces the validation efforts, which can take about 70% of the overall design cost. The abstract level at which the design is addressed (in the synchronous approach) enables to deal with the inherent complexity of distributed systems by leaving away the unnecessary details. It therefore favors a rapid exploration of different alternatives about system distribution so as to get relevant feedback for an adequate design choice.

However, in order to take advantage of all the above features of our approach in the design of real-life systems, the designer must have some knowledge of synchronous programming. In order to overcome this obstacle, there are currently significant efforts to provide a very advanced user-friendly interface for POLYCHRONY. The resulting new environment, referred to as SME, is an Eclipse plugin. It aims to be accessible to a wide
audience and is available online for free at the same address as POLYCHRONY. On the other hand, in the current status of our approach, some expertise in the SIGNAL clock calculus may be required to check the clock hierarchies, which serve to characterize the correctness of a system distribution. An important future improvement of our work is the automatic application of the distribution criteria (endo-isochrony) on any SIGNAL model of a system so as to decide about its partitioning. Finding a solution to this issue will save the designer from analyzing possible complex clock hierarchies generated by the compiler for the distribution. Finally, for real systems, we can also mention another limitation of our approach, resulting from the simplifications considered in the architecture model. All physical components such as memory caches or hardware accelerators and their specific characteristics are not taken into account. Some abstractions of these components could be studied, and taken into account to ameliorate the distribution criteria.

7 Conclusions
In this paper, we presented how safety-critical distributed embedded systems are practically designed using the polychronous model, associated with the synchronous language SIGNAL. This model enables to describe systems with components that hold different activation clocks. We proposed a methodology that ensures a correct-by-construction implementation of these systems from high-level models in POLYCHRONY, the development framework of SIGNAL. This has been demonstrated on a case study from the avionic domain. The endochrony and endo-isochrony properties of SIGNAL specifications have been used as key notions to design the distribution of applications on execution platforms. These platforms contain asynchronous communication mechanisms, which have been also modeled in SIGNAL and proved to achieve message exchanges correctly. An important remark is that the formal verification techniques and tools available in POLYCHRONY are applicable at almost all design steps of our methodology in order to check the correctness of intermediate results. In particular, the analysis of non functional properties of the deployed system model has been addressed by dealing with clock synchronizability issues between the different multirate parts of the system. Finally, a distributed code can be automatically generated from this model.

A very interesting perspective to the presented work concerns the automatic application of the identified design criteria in SIGNAL to distribute a system. That is, given a system \( S \), how could the compiler automatically use the endo-isochrony property to distinguish the different sub-parts \( S_1, \ldots, S_n \) to be mapped on different processors such that \( S = S_1 \mid \ldots \mid S_n \), and how could it select the suitable communication mechanisms from an existing library? It seems that the answer to these questions, in particular to the former, requires to analyze the combinatorics of the solution space of a system decomposition, which can be a priori large.

References
messages of the queue. The semantics of this operator is such that every reading of value on the input flow leads to a sliding on the left of the values already present in the defined array. The new value is therefore inserted on the right. The following trace illustrates such a scenario on a 2-sized queue. The symbol "\( \ast \)" designates a given initialization value.

\[
\text{msg}_\text{in} : \begin{cases} \text{queue} \rightarrow \text{[\( \ast, \ast \)]} \text{ [\( \ast, 1 \)]} \text{ [\( 1, 2 \)]} \text{ [\( 2, 3 \)]} \text{ [\( 3, 4 \)]} & \ldots \end{cases}
\]

For the equation: \( \text{queue} := \text{new}_\text{msg}_\text{in} \text{ in window size} \), the semantics of the sliding window yields (for the sake of simplicity, initial values are omitted): \( \forall t \geq 0 \)

\[
(t + i \geq \text{size}) \Rightarrow (\text{queue}_\text{[i]} = \text{new}_\text{msg}_\text{in}_{\text{int}+\text{size}+i+1})
\]

Thanks to this semantics of the window operator, the values that are taken into account in the flow associated with the input signal \( \text{msg}_\text{in} \) are put in queue according to their occurrence order. Thus, the property \((E_1)\) holds by definition.

**Proof of \((E_2)\).** The output flow is determined by the equation defined at lines 11 and 12 in Fig.\[5\] in safe_FIFO, only the second argument of the default operator has to be considered since the first one is forbidden by Property \((E_1)\). On a read request, we denote by \( i \) the index of the message retrieved from the array \( \text{queue} \). The value of \( i \) is given by the difference between the maximal size (i.e. \( \text{size} \)) of the message queue and the previous number of messages, \( \text{prev}_{\text{nbmsg}} \) of the queue: \( \forall t \geq 0 \), \( i = \text{fifo}_{\text{size}} - \text{prev}_{\text{nbmsg}} \).

Let us consider the following invariant: at any logical instant \( t \), the index \( i_t \) indicates the oldest message enqueued when the message queue is not empty. We need to check that this invariant is always preserved after any access to the queue.

1) Case 1: \( \text{read at an instant} \ k \). When a message is read at an instant \( k \), the signal \( \text{nbmsg} \) is decremented by one (equation at lines 5 and 6 in Fig.\[5\], i.e.: \( \text{nbmsg}_{k+1} = \text{prev}_{\text{nbmsg}}_{k} - 1 \).
At instant \( k+1 \), we have: \( \text{prev}_{\text{nbmsg}_{k+1}} = \text{nbmsg}_{k} \). Thus,

\[
i_{k+1} = \text{fifo}_{\text{size}} - \text{prev}_{\text{nbmsg}_{k+1}}
\]

\[
i_{k+1} = \text{fifo}_{\text{size}} - \text{nbmsg}_{k}
\]

\[
i_{k+1} = \text{fifo}_{\text{size}} - \text{prev}_{\text{nbmsg}_{k}} - 1
\]

\[
i_{k+1} = (\text{fifo}_{\text{size}} - \text{prev}_{\text{nbmsg}_{k}}) + 1
\]

\[
i_{k+1} = i_{k} + 1
\]

On the other hand, on each read request, the array represented by the signal \( \text{queue} \) remains unmodified. As a result, the invariant is preserved since the oldest element in the message queue becomes the one with index \( i_{k+1} \) in the array.

2) Case 2: \( \text{write at an instant} \ k \). When a message is effectively written in the message queue at an instant \( k \), the signal \( \text{nbmsg} \) is incremented by one (equation at lines 5 and 6 in Fig.\[5\], i.e.: \( \text{nbmsg}_{k} = \text{prev}_{\text{nbmsg}}_{k} + 1 \). At instant \( k+1 \), we have: \( \text{prev}_{\text{nbmsg}_{k+1}} = \text{nbmsg}_{k} \). Thus, similarly to the previous case, we obtain that: \( i_{k+1} = i_{k} - 1 \).

Contrarily to Case 1, here the signal \( \text{queue} \) is modified. As a matter of fact, the sliding window on the signal \( \text{new}_\text{msg}_\text{in} \) will take into account the occurrence of this signal at instant \( k \). The elements that are already present in queue are shifted according to the semantics of the window operator (i.e. a shift of one cell towards lower indices of the array and insertion of the new message in the cell with the highest index value). Hence, the index \( i_{k+1} \) always corresponds to the same message as the previous instant. So, the invariant is preserved.