Asynchronous-Channels Within Petri Net-Based GALS Distributed Embedded Systems Modeling

Filipe Moutinho and Luís Gomes, *Senior Member, IEEE*

***Abstract*—Model-based development approaches can provide a major contribution in the development of globally asynchronous locally synchronous distributed embedded systems (GALS-DES) if supported by suited modeling formalisms and design automation tools. The use of Petri nets (either low-level or high-level classes) extended with asynchronous-channels (ACs), time domains, priorities, inputs, and outputs is proposed in this paper to model GALS-DES (composed by deterministic components), ensuring that the created GALS model is locally deterministic, distributable, network-independent, and platform-independent. The proposed ACs, with high level of abstraction, specify the components interaction through Petri net objects with specific attributes that unambiguously identify this interaction within the GALS model. Two algorithms are proposed to translate and decompose the GALS model into Petri net models without ACs, which can be used as inputs in model-checking tools and automatic code generators supporting GALS-DES verification and implementation. The specification of a small goods lift distributed controller illustrates the use of the proposed ACs.**

***Index Terms*—Asynchronous channels, distributed embedded systems (DESs), globally asynchronous locally synchronous (GALS) systems, model-based development (MBD), Petri nets.**

I. Introduction

DISTRIBUTED embedded systems (DESs), networked embedded systems, or cyber physical systems (CPSs) are sets of computer control systems (often embedded in devices, machines, or infrastructures) in interaction, performing specific tasks, usually to improve our quality of life. A single automation system or embedded system can also be understood as a distributed system, if composed by a set of components in interaction.

The implementation of an automation system or embedded system as a network of distributed components and sensors (rather than being implemented as a standalone system) may lead to higher performance, lower power consumption, lower electromagnetic interference (EMI), and lower production costs. This is justified, because each part of the system can be implemented in the most suited platform/device, working at an optimized clock frequency (to reduce power consumption and EMI). Several criteria, such as system performance, power consumption, EMI, and cost, can be considered to select the most appropriate implementation platforms, which may be composed by software and hardware components. Additionally, the implementation of a system using a set of components may allow the reuse of previously developed components, reducing the development time and the production costs. However, distributed systems tend to be more complex than centralized systems, due to the complexity introduced by the components interaction [1].

Model-based development (MBD) or model-driven development approaches, as the model-driven architecture (MDA) initiative defined by the Object Management Group (OMG) [2], have been proposed to develop software systems. MBD approaches, such as those presented in [3]–[10], have also been proposed to develop industrial control systems, embedded systems, and cyber physical system (CPS), resulting in systems better documented, developed in less time, with less development errors, and benefiting from reusability of models to produce new code for different platforms. They can also improve the interaction between customers, systems analysts, and the development team, allowing the development of improved embedded systems.

Several graphical modeling formalisms, such as finite state machines, Statecharts [11], Petri nets [12]–[14], unified modeling language (UML)-modeling and analysis of real-time and embedded (MARTE) systems [2], and systems modeling language (SysML) [2], have been successfully used to develop computer control systems using MBD approaches. Among these formalisms, we highlight Petri nets, which are a graphical formalism with the capability to explicitly specify parallelism, concurrency, and synchronization (usual features of automation and embedded systems). Petri nets have a strong mathematical definition and a well-defined semantics, allowing the use of verification and automatic code generation tools [14] to achieve the mentioned MBD benefits. The international standards International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 15909-1 and 15909-2 define low-level and high-level Petri nets (HLPNs) [15]. Several autonomous Petri net classes have the capability of specifying not only the controller but also its interaction with the environment, namely the synchronized Petri nets (SPNs) [16], the net condition/event systems (NCESs) [17], [18], the signal interpreted Petri nets

Manuscript received May 31, 2013; revised June 06, 2014; accepted June 28, 2014. Date of publication July 23, 2014; date of current version November 04, 2014. This work was supported by Portuguese Agency Fundação para a Ciência e a Tecnologia (FCT) in the framework of project PTDC/EEI-AUT/2641/2012. The work of F. Moutinho was supported by FCT Grant SFRH/BD/62171/2009. Paper no. TII-14-0230.

The authors are with the Faculty of Sciences and Technology, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal; and also with the Center of Technology and Systems, Instituto de Desenvolvimento de Novas Tecnologias (UNINOVA), 2829-516 Caparica, Portugal (e-mail: fcm@uninova.pt; lugo@fct.unl.pt).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2014.2341933

[19], and the input–output place-transition (IOPT) Petri net class (IOPT net) [20].

Globally asynchronous locally synchronous (GALS) systems, initially proposed in [21], are composed by a set of synchronous components in interaction, bringing together the benefits of synchronous and asynchronous systems. Despite the possible advantages and interest of the scientific community, the application of GALS techniques is rare [22]. The lack of computer-aided design (CAD) tools to automate their development is pointed in [22] as a reason that may justify the low interest in GALS techniques by the industry. Krstić *et al.* [23] argue that the strength of GALS systems is based on its design methods that will support the implementation of large systems-on-chip (SoCs) in the future. A GALS system is often understood as a SoC, but it can also be a system-off-chip, such as in [24], where a GALS model was proposed for the IEC 61499 international standard. In this paper, the term GALS DES is understood as a distributed system (geographically distributed, in a single implementation platform, or in a single chip), composed by a set of synchronous components (which can include hardware and software components), where each component is synchronous with a specific clock signal.

To support an MBD approach for GALS-DES, Petri nets with time domains, priorities, inputs, and outputs are extended in this paper with three types of generic asynchronous channels (ACs). This MBD approach is based on the network- and platform-independent specification of GALS-DES through Petri net models, which support their simulation (using simulation tools), the verification (using model-checking tools), and the components and communication channels implementation (using automatic code generators). The use of network- and platform-independent models allows a later decision about the implementation platforms and communication networks, enabling the implementation in the most suited platforms, with the most suited communication networks, to obtain the required power consumption, performance, EMI, and platform costs. Additionally, these platform-independent Petri net models support the automatic code generation of software programming languages and hardware description languages, to be deployed into heterogeneous implementation platforms.

The proposed ACs, which are represented by Petri net objects with specific attributes, specify the interaction between synchronous components (specified by Petri net submodels). The ACs, with high-level of abstraction, abstract any type of communication network. The use of Petri nets extended with these channels, leads to specifications where the focus is on the components specification, on what triggers their interaction, and on the consequences of that interaction, and not on how they communicate. The simulation and verification of these specifications provide behavioral proprieties that exist in the GALS-DES whatever their communication networks are. The verification of these specifications additionally provides data to scale the memory resources of the components and of the communication nodes, supporting the automatic generation of the GALS-DES implementation code. It is important to note that the use of Petri net classes extended with the proposed channels and with time domains and priorities, ensure that the resulting GALS models are locally deterministic, distributable, network-, and platform-independent and that identify what is components interaction and what is components computation, avoiding ambiguities both by modelers and design automation tools.

The advantages of specifying GALS-DES through Petri nets extended with generic ACs rather than specifying GALS-DES through Petri nets without these channels are presented in Section II. Also in Section II, the proposed ACs are compared with the existing communication channels. Section III briefly presents the concept of time domain, how it adds the GALS execution semantics into Petri nets, and how priorities can be used to solve conflicts and behavioral ambiguities. Three generic ACs, for (low-level and high-level) Petri nets, are defined in Section IV, to support the specification in a generic way of the asynchronous interaction between the synchronous components of GALS-DES. The behavior of the GALS-DES specifications can then be verified using model-checking tools, which also provide information about the required resources to implement the systems. To enable the use of a model-checking tool, a translation algorithm is proposed in Section IV. Section V describes how the specification can be automatically decomposed into a set of implementable submodels, supporting the implementation through automatic code generators. A tool chain framework [25] (http://gres.uninova.pt/) was extended and used to develop GALS-DES. One of those systems (the distributed controller of a small goods lift) is presented in Section VI, illustrating the defined channels application. Finally, Section VII presents the conclusion.

II. Related Work

A. Petri Nets Supporting GALS Systems Modeling

Most Petri net classes are suited to specify distributed systems; however, just a few support the specification of GALS systems. Using SPNs [16], NCESs [17], [18], signal net systems (SNSs) [26], [27], distributed timed-arc Petri nets (DTAPNs) [28], place/transition-nets with localities (PTL-nets) [29], elementary net systems with localities (ENL-systems) [30], or the IOPT nets [20] extended for GALS systems [31], it is possible to specify GALS systems.

In SPNs [16], each transition is synchronized by an external event. Given that each external event can synchronize a set of transitions and different external events are independent, SPNs can be used to create GALS models. The interaction between synchronized submodels can be specified through additional submodels.

NCESs and SNSs, being nontimed or timed, also support the specification of GALS systems. To specify synchronous components using nontimed models, “greedy” transitions connected to sets of transitions through event signals are used to create sets of synchronized transitions (forcing these transitions to fire), as presented in [26]. To specify synchronous components using timed models, “synchronization sets” can be used to create sets of synchronized transitions [26]. Given that, the communication channels that support the interaction between components can also be specified through (NCES or

SNS) submodels, the global GALS systems can be specified through NCES or SNS models.

DTAPNs [28] can support the specification of GALS systems; however, this Petri net class is not appropriate to specify deterministic components. In DTAPN, places can have different clock rates and tokens have associated ages, which are 0 in the initial marking and when they are added to places, becoming older at their places clock rates. The arcs connecting places to transitions may have an associate annotation with a time interval, stating the age of the tokens that can enable the transition. Given that enabled transitions of DTAPN may fire or not (are not synchronized), this Petri net class is nondeterministic, making it unsuitable to specify the deterministic components of GALS systems.

PTL-nets [29] and ENL-systems [30] have been proposed to specify GALS systems. These Petri net classes associate the notion of locality to transitions, creating Petri net models with sets of synchronous transitions (specifying synchronous components). The communication between components is specified through buffer places (BPs). But BPs do not support, for instance, the specification of communication channels with undefined communication delays. Additionally, these two Petri net classes (such as the DTAPN) are autonomous, not supporting the specification of the controllers' inputs and outputs.

Also to support the specification of GALS systems, the IOPT net class [20] was extended in [31]. The time-domain concept was proposed to associate each place and each transition with a specific synchronous submodel that specifies a synchronous component. To specify the communication between those Petri net submodels, one type of AC was presented. The use of this channel, when compared to the use of BPs (used in [29], [30] to specify components interaction), has the advantage of providing a network independent specification. This specification supports the verification and provides high flexibility in the implementation phase, supporting the use of the most suited communication channels, network protocols, and network topologies, to interconnect the distributed synchronous components.

The specification of GALS-DES through a Petri net-based formalism extended with ACs and time domains (TDs), when compared to their specification through SPNs, NCESs, SNSs, PTL-nets, or ENL-systems, has some advantages, as presented in Table I. The main advantages are the use of a formalism, which besides ensuring that the created models are GALS, it also ensures that the created models are always distributable, network-independent, and free of structural ambiguities (regardless of specifying the desired behavior). These models are always distributable, because the time-domain concept together with the proposed ACs ensures that all normal places and transitions belong to well delimited synchronized domains, and two synchronized domains (two subnets with different time domains that specify two different components) can only be connected through ACs, ensuring that those subnets cannot be connected, for instance, through a normal arc or an event signal [17] that would make the model not distributable (since it does not specify the asynchronous interaction). Given that, the proposed ACs specify the interaction without providing any specific details about how the messages are exchanged and about the communication delay, they ensure network-independent specifications. Additionally, these ACs abstract the components' interaction through Petri net objects with specific annotations or attributes that are graphically represented through text labels and/or through Petri net objects with specific shapes, ensuring that the communication and the computation are unambiguously identified in the models (which is not true when communication is specified with submodels without specific annotations or attributes). This enables, without requiring a later identification on which are the channel submodels and the component submodels, the use of automatic code generator tools to generate the communication nodes implementation code and the components implementation code.

TABLE I
ADVANTAGES OF USING PETRI NETS EXTENDED WITH ACS AND TDS IN THE DEVELOPMENT OF GALS-DES

	Petri net classes	
Advantages	SPNs [16], NCESs [17], SNSs [26], PTL [29], or ENL [30]	Petri nets with ACs and TDs
Are always distributable	No	Yes
Are always network independent	No	Yes
Are always free of structural ambiguities	No	Yes

Specifying the interaction using the proposed ACs, rather than through submodels without specific annotations or attributes, can reduce the global GALS-DES model size, as illustrated in Fig. 1. Fig. 1(a) model with ACs has 18 nodes, whereas Fig. 1(b) model (where the ACs were replaced by their behaviorally equivalent models) has 36 nodes (the reference nodes were not counted). However, this additional advantage was not the main motivation to define the proposed channels.

It is important to note that when using low-level Petri nets, such as SPNs, PTL-nets, or Petri nets with ACs and TDs, to ensure locally determinism, it is not sufficient to have all transitions synchronized, it is also required to ensure that the created models are free from effective conflicts. To solve conflicts, transitions' priorities [16] can be used.

An overview of the channels that were proposed for Petri nets is presented in Section II-B. Most of those channels are inappropriate to specify the exchange of messages between GALS systems' components through network communication nodes, or do not provide network-independent specifications. Finally, the ACs proposed in this paper are compared with other ACs, such as the one presented in [31].

B. Communication Channels in Petri Nets

This section presents an overview of the existing communication channels for Petri nets, and compares the existing channels with the ones defined in this paper, as summarized in Table II. The presented survey summary refers to papers proposing or using channels to specify the interaction between Petri net submodels, in order to develop (software and

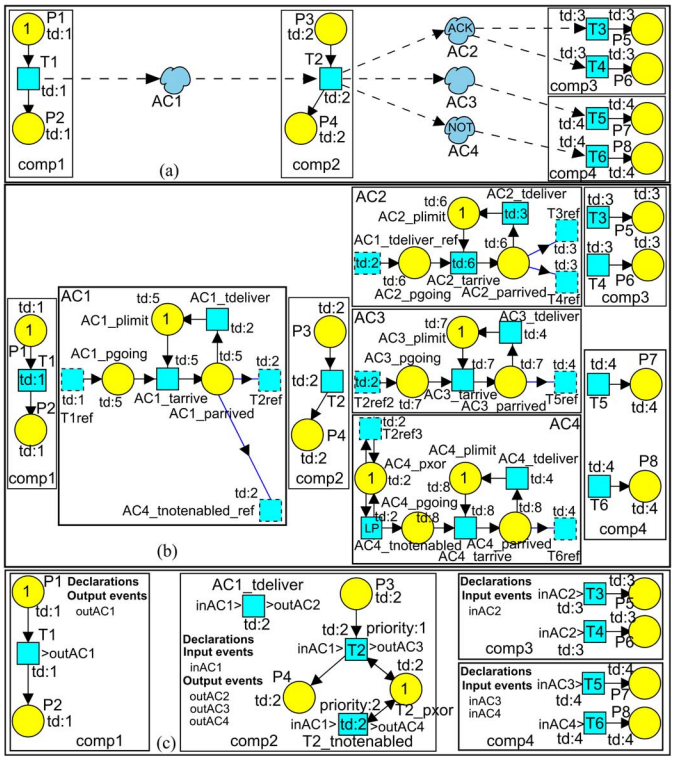

Fig. 1. (a) Global model of a distributed system. (b) Model that presents the execution semantics of the (a) model. (c) Submodels that support the components implementation of the (a) model.

TABLE II
ADVANTAGES OF USING THE PROPOSED ACS

	Channels				
Advantages	SCs	BPs	ACs [31], [39]	ACs [40]	New ACs
Ensure network independent models	No	No	Yes	No	Yes
Can have several target transitions	No/ Yes	No/ Yes	No	Yes	Yes

hardware) systems. It excludes papers and works about the use of Petri nets to model and analyze specific communication protocols, as in [32] and [33]. The channels are classified in this section as symmetrical or asymmetrical (directed) and as synchronous or asynchronous.

Synchronous channels (SCs, symmetrical or asymmetrical) are used in Petri nets to specify the interaction between transitions, which may belong to different Petri net submodels (modules), considering communications with zero-time delay. Probably the most well-known communication channel for Petri nets is the symmetrical and SC proposed in [34]. When two transitions are connected through a symmetrical SC, both transitions must be simultaneously enabled to fire, and they will always fire atomically at the same time instant. The event signals from NCESSs [17], [18] and SNSs [27] can be called asymmetric (directed) SCs, such as those proposed in [35] and [36]. When two transitions are connected through an asymmetrical (directed) SC, one transition is the source and the other is the target. When the source fires, the target also fires (if enabled) at the same time instant. These channels do not support the specification of communications with delays (which may be unknown) and through unreliable channels (channels that may not ensure the integrity of the message or may loose it).

Several papers, such as in [37], [38], and [29], propose the use of BPs to specify the asynchronous interaction between submodels. These buffers are suited to specify interactions through shared variables (after sending a message, it is immediately available to the receiver), but unappropriated to specify the exchange of messages, for instance, through unreliable communication channels or through networks with variable communication delays, where a message is only available to the receiver after a variable amount of time.

To specify the exchange of messages (between distributed synchronous components) through communication channels that introduce delays (common in real communication scenarios), an AC was introduced in [31]. This channel can be used to connect transitions from the source component to transitions of the target component, and have the following execution semantics: when the source transition fires, an event is sent to the target transition, which will fire (if enabled) in a future time instant. This channel, which specifies communications with delays, is suited to specify the interaction through unreliable channels or through channels with uncertain communication time.

An improved version of the channel introduced in [31] is defined in Section IV. The new channel, named as simple AC (SimpleAC), is defined as a subnet with a special type of place connected to transitions using a special type of arc (instead of being defined as a Petri net page annotation). Among the differences between the new channel and the previous one we highlight the possibility to have a set of target transitions instead of just one.

To support the specification of the interaction between submodels obtained after a model transformation, an asynchronous channel was introduced in [39]. This channel sends messages from a source transition to a target transition and additionally returns to the source component messages with feedback about the target transition firing. The main disadvantage of this channel, such as in the channel proposed in [31], is that it can only have one target transition. The behavior specified by this channel can be specified through the channels defined in Section IV, using two SimpleACs and one Not-enabled AC, but with the advantage of being able to be connected to multiple target transitions.

Three ACs were introduced in [40] for HLPNs, supporting the specification of data transmission. The disadvantage of these channels is their first in, first out (FIFO) semantics, which restrict their application to the specification of communication networks that ensure the order of the messages, not being network-independent as desired in the current paper. The channels proposed in Section IV do not specify the order of the messages, as desired.

III. TIME DOMAINS AND PRIORITIES

The ACs defined in this paper support the specification of the asynchronous interaction between synchronous

components of distributed GALS systems. A GALS system is composed by a set of synchronous components in interaction, where each component has a specific execution step, (time domain), which is independent from other components execution steps (with distinct time domains). In a hardware component, the execution step is often given by its clock signal. The time-domain concept was proposed in [31] and introduce the GALS execution semantics into Petri nets, enabling the specification of GALS systems components using Petri nets and ensuring that the created GALS models are distributable.

A Petri net class that includes the concept of time-domain is a tuple that should include at least three sets and two functions: PN = $(P, T, A, \text{priority}, \text{td})$. P is a finite set of places, T is a finite set of transitions, and A is a set of arcs, such that $A \subseteq ((P \times T) \cup (T \times P))$ and $\forall_{(n_1 \times n_2) \in A}(\text{td}(n_1) = \text{td}(n_2))$. Priority is a partial function that applies transitions to positive integers: priority: $T' \rightarrow \mathbb{N}$, such that any two transitions with the same source place must have different priorities (the one with lower value has higher priority), solving conflicts. td is a function that applies Petri net nodes (places and transitions) to positive integers that identify the associated components: td: $(P \cup T) \rightarrow \mathbb{N}$.

Time domains make Petri nets totally synchronized, with single-server semantics, and with well delimited synchronized domains. All transitions with a specific time domain are synchronized by an implicit external event, making the model totally synchronized (totally SPNs were proposed in [16] but with explicit external events). This means that the transitions that are enabled when the associated event occurs and not in conflict (not in an effective conflict), will fire simultaneously at that instant. All effective conflicts are solved by priorities. Additionally, each transition can only fire once at each execution step (single-server semantics [41]). All nodes with a specific time domain belong to the submodels (which are well delimited, making them distributable) of a specific synchronous component. Given that, in real implementations, two actions from two distributed components (which are synchronized by different clock signals) never fire exactly at the same time instant, it was defined that two transitions with different time domains never fire simultaneously. The time-domain concept do not provide any information about the execution frequencies (ensuring platform-independent specifications), which are defined in a later stage of the development flow.

Time domains' and transitions' priorities (given by the priority function) make low-level Petri net models locally deterministic, but are insufficient to ensure that HLPN models are locally deterministic. To ensure locally determinism in HLPNs, it is additionally required to avoid ambiguities (if several bindings of tokens can enable a transition, when it fires which tokens are destroyed?). To avoid HLPNs ambiguities, several rules can be considered [42], setting tokens' priorities.

IV. Asynchronous Channels

To specify the asynchronous interaction (the exchange of messages) between GALS distributed components specified by Petri nets with time domains, three types of ACs are proposed: 1) simple AC (SimpleAC); 2) acknowledged AC (AckAC); 3) not-enabled AC (NotAC).

This section presents these channels' definition and execution semantics for low-level and HLPN classes. When used in high-level classes, ACs have extra annotations, specifying the data variables that are transmitted by the channel.

Each AC specifies message sending from one transition (source transition of the source component with a specific time domain) to a set of transitions (target transitions of the target component with another specific time domain). A SimpleAC sends a message to the target transitions each time the source transition fires; an AckAC sends a report message (acknowledged) to the target transitions each time the source transition receives a message; and a NotAC sends a report message to the target transitions each time the source transition receives a message and does not fire (not enabled). There is a delay between the moment when the message leaves the source component and the moment when it arrives into the target component. The message is then replicated and simultaneously delivered to all its target transitions. Only the target transitions that are enabled (when the message is delivered) fire.

An AC is a subnet composed by an *asynchronous channel place*, connected to transitions using *asynchronous channel arcs* (one *source channel arc* and one or more *target channel arcs*). An *asynchronous channel place* is a (special type of) place graphically represented by a cloud, which has the inscription "ACK" if it is an AckAC, the inscription "NOT" if it is a NotAC, or without inscription if it is a SimpleAC. *Asynchronous channel arcs* are the special types of arcs graphically represented by dashed arrows.

Fig. 1(a) presents a simple Petri net model with two SimpleACs (AC1 and AC3), one AckAC (AC2), and one NotAC (AC4), specifying the interaction between four distributed components. When transition $T1$ fires a message, it is sent through AC1 to $T2$, which then sends a new message (an acknowledged) through AC2 to $T3$ and $T4$. Additionally, when the message is delivered to $T2$, which is not enabled (it does not fire), another message (a not-enabled report) is sent through AC4 to $T6$. If place *P3* was marked, then $T2$ would fire, and instead of being sent a message through AC4, a message through AC3 (to $T5$) would be sent.

A. Definition

A set of ACs is given by AC $= (P_{\text{ac}}, A_s, A_t)$, where P_{ac} is a set of *asynchronous channel places*, such that $P_{\text{ac}} = (P_{\text{sac}} \cup P_{\text{aac}} \cup P_{\text{nac}})$ and $P_{\text{sac}} \cap P_{\text{aac}} \cap P_{\text{nac}} = \emptyset$, $A_s = (T_s \times P_{\text{ac}})$ is a set of *source channel arcs* connecting transitions ($T_s \subseteq T$) to *asynchronous channel places*, and $A_t = (P_{\text{ac}} \times T_t)$ is a set of *target channel arcs* connecting *asynchronous channel places* to transitions ($T_t \subseteq T$). Each AC has one *asynchronous channel place* ($\#P_{\text{ac}} = \#\text{AC}$) that: 1) has one and only one input arc, which is a *source channel arc*: $\forall_{p_{\text{ac}} \in P_{\text{ac}}}(\exists!_{a_s \in A_s} : a_s = t_s \times p_{\text{ac}})$; and 2) has one or more output arcs (*target channel arcs*): $\forall_{p_{\text{ac}} \in P_{\text{ac}}}(\exists_{a_t \in A_t} : a_t = p_{\text{ac}} \times t_t)$. All target transitions of an AC must have equal time-domains: $\forall_{(p_{\text{ac}} \times t_1),(p_{\text{ac}} \times t_2) \in A_t}(\text{td}(t_1) = \text{td}(t_2))$. One transition cannot be target of two ACs: $\forall_{(p_{\text{ac}1} \times t_1),(p_{\text{ac}2} \times t_2) \in A_t}(p_{\text{ac}1} \neq p_{\text{ac}2} \Rightarrow t_1 \neq t_2)$ (which will allow a much more simple execution

semantics definition for ACs). In the unlikely modeling situation that one wants to consider one transition as target of two ACs, it will be always possible to duplicate the target transition obtaining similar results. The source transition time domain of an AC must be different from the time domain of its target transitions: $\forall_{(t_s \times p_{\mathrm{ac1}}) \in A_s} \forall_{(p_{\mathrm{ac2}} \times t_t) \in A_t} (p_{\mathrm{ac1}} = p_{\mathrm{ac2}} \Rightarrow \mathrm{td}(t_s) \neq \mathrm{td}(t_t))$. If the AC is an AckAC or a NotAC, its source transition must be the target transition of another AC: $\forall_{(t_s \times p_x) \in A_s} (p_x \in P_{\mathrm{aac}} \vee p_x \in P_{\mathrm{nac}}) \exists!_{(p_{\mathrm{ac}} \times t_t) \in A_t} (t_t = t_s)$.

When used in HLPNs, $\mathrm{AC} = (P_{\mathrm{ac}}, A_s, A_t, \mathrm{cv})$, where $cv : A'_s \Rightarrow \mathcal{P}(V)$ is a partial function applying *source channel arcs* to subsets of sorted variables (V), specifying the data transmitted from the source into the target transitions. V is defined in the ISO/IEC 15909. Each AckAC or NotAC can only send variables that were received by its source transition (coming from the associated AC).

B. Semantics

The behavior of any Petri net model with ACs, such as the one presented in Fig. 1(a), can be specified through an equivalent Petri net model without ACs, which presents the semantics of the initial model, as illustrated in Fig. 1(b). Although Fig. 1(b) shows a set of disconnected subnets, it presents a single model. To the merge the subnets, the reference transitions [15] [represented in Fig. 1(b) by dashed squares] must be merged with the transitions that they refer. Fig. 2 presents the translation algorithm that reads the global model and creates a new model where each AC is replaced by its equivalent subnet. As presented in Fig. 2, the *SimpleAC* and the *AckAC* have equal equivalent subnets [as illustrated in Fig. 1(b)]; however, the *SimpleAC* equivalent subnet is connected to its source transition, whereas the *AckAC* equivalent subnet is connected to the transition *tdeliver* of the AC that is the source of its source transition. The *NotAC* equivalent subnet extends the *SimpleAC* equivalent subnet [as illustrated in Fig. 1(b)].

```
1: nPN ← gPN ← Load(globalPNname)
2: for all p_ac ∈ nPN.P_ac do
3:     a_s : a_s ∈ nPN.A_s ∧ a_s = (t_s, p_ac)
4:     nPN.AddNewPlace(pgoing, cv(a_s))
5:     nPN.AddNewPlace(parrived, cv(a_s))
6:     nPN.AddNewPlace(plimit, marking = 1)
7:     nPN.AddNewTransition(tarrive, UNIQUE_TD)
8:     nPN.AddNewTransition(tdeliver, nPN.td(p_ac•))
9:     nPN.AddNewArc(pgoing, tarrive, cv(a_s))
10:    nPN.AddNewArc(tarrive, parrived, cv(a_s))
11:    nPN.AddNewArc(parrived, tdeliver, cv(a_s))
12:    nPN.AddNewArc(tdeliver, plimit)
13:    nPN.AddNewArc(plimit, tarrive)
14:    if p_ac ∈ nPN.P_nac then
15:        t_s : a_s = (t_s, p_ac)
16:        nPN.AddNewTransition(tnotenabled, nPN.td(t_s))
17:        nPN.AddNewPlace(pxor, marking = 1)
18:        nPN.AddTwoNewArcs(t_s, pxor)
19:        nPN.AddTwoNewArcs(tnotenabled, pxor)
20:        nPN.AddNewArc(tnotenabled, pgpoing, cv(a_s))
21:        nPN.AddNewPriorityHigherLower(t_s, tnotenabled)
22:    end if
23:    nPN.RemovePlace(p_ac)
24: end for
25: for all a_s ∈ gPN.A_s : a_s = (t_s, p_ac) do
26:    if p_ac ∈ gPN.P_sac then
27:        nPN.AddNewArc(t_s, pgoing, cv(a_s), , p_ac)
28:    end if
29:    p_acs : (a_t ∈ gPN.A_t ∧ a_t = (p_acs, t_t) ∧ t_t = t_s)
30:    if p_ac ∈ gPN.P_aac then
31:        nPN.AddNewArc(tdeliver, pgoing, cv(a_s), p_acs, p_ac)
32:    end if
33:    if p_ac ∈ gPN.P_nac then
34:        a_x : (a_x ∈ gPN.A_s ∧ a_x = (t_x, p_ac1) ∧ p_ac1 = p_acs)
35:        nPN.AddNewTA(parrived, tnotenabled, cv(a_x), p_acs, p_ac)
36:    end if
37:    nPN.RemoveArc(a_s)
38: end for
39: for all a_t ∈ nPN.A_t : a_t = (p_ac, t_t) do
40:    a_s : (a_s ∈ gPN.A_s ∧ a_s = (t_s, p_ac1) ∧ p_ac1 = p_ac)
41:    nPN.AddNewTestArc(parrived, t_t, cv(a_s), p_ac, )
42:    nPN.RemoveArc(a_t)
43: end for
44: CreateNewPNML(nPN)
```

Fig. 2. Translation algorithm that reads the global model and creates the equivalent model without ACs.

V. Verification and Implementation of GALS-DES

The defined channels are proposed not only to enable the distributed system specification using Petri nets, but also to support its verification and implementation. Since it is possible to transform a Petri net model with ACs into a Petri net model without channels and with the same behavior, the resulting Petri net model can be used as input in model-checking tools that do not know the AC concept. State-space-based model-checking tools provide model proprieties' verification, supporting behavioral analysis and data about the required resources to implement the distributed system. The model-checking tool [25], which was extended to support the state-space generation of models with time domains, supports the search for proprieties (in the state space, also known as reachability tree), which can be expressed in computation tree logic (CTL).

After model verification and before its implementation as a set of distributed components, it is required to decompose the model with ACs into a set of implementable submodels, specifying the distributed components. To create the implementable submodel of one specific component, the decomposition algorithm presented in Fig. 3 makes a copy of the global model, removes the objects and annotations that do not specify that component, and inserts extra subnets and input and output events to specify the interaction between the component and the communication nodes. The four submodules presented in Fig. 1(c) were created using this algorithm.

Input and output events (IE and OE) are used to specify the interaction between the components and the communication nodes, and can also be used to specify the interaction between the components and the environment. Events are associated to Petri net transitions, given by the partial functions $\mathrm{ie} : T' \rightarrow \mathcal{P}(\mathrm{IE})$ and $\mathrm{oe} : \mathrm{OE} \rightarrow T$. When used in HLPNs, events can have associated variables (V) given by iev : IE' $\rightarrow \mathcal{P}(V)$ and oev : OE' $\rightarrow \mathcal{P}(V)$.

VI. Application Example

To validate the defined channels, the IOPT net class [20], [31] and its tool chain framework [25] (http://gres.uninova.pt/) were extended with the proposed ACs and used to develop GALS-DES. The extended framework supports GALS-DES specification (using a model editor), verification (using a

```
1: gPN ← Load(globalPNname)
2: tdList ← GetTds(gPN)
3: for td ∈ tdList do
4:    nPN ← gPN
5:    for all p ∈ nPN.P do
6:       if nPN.td(p) ≠ td then
7:          nPN.RemovePlace(p)
8:       end if
9:    end for
10:   for all p_ac ∈ nPN.P_ac do
11:      if ∃(a_t ∈ gPN.A_t) : a_t = (p_ac, t_t) ∧ td(t_t) = td then
12:         a_s : a_s ∈ gPN.A_s ∧ a_s = (t_s, p_ac)
13:         nPN.AddNewInEv(p_ac, cv(a_s))
14:      end if
15:      if ∃(a_s ∈ gPN.A_s) : a_s = (t_s, p_ac) ∧ td(t_s) = td then
16:         nPN.AddNewOutEv(p_ac, cv(a_s))
17:      end if
18:      if targetIsAckACsource(p_ac, gPN) then
19:         inEvRef ← CreateInEvRef(p_ac, gPN)
20:         outEvRefs ← CreateOutEvRefs(p_ac, gPN)
21:         nPN.AddNewT(tdeliver, td, inEvRef, outEvRefs)
22:      end if
23:      nPN.RemovePlace(p_ac)
24:   end for
25:   for all t ∈ nPN.T do
26:      if nPN.td(t) ≠ td then
27:         nPN.RemoveTransition(t)
28:      else
29:         isNACsource ← FALSE
30:         inEvRef ← NULL
31:         outEvRefs ← emptyList()
32:         for all p_ac ∈ gPN.P_ac do
33:            if ∃(a_t ∈ gPN.A_t) : a_t = (p_ac, t) then
34:               inEvRef ← CreateInEvRef(p_ac)
35:               t.AddInEvRef(inEvRef)
36:            end if
37:            if ∃(a_s ∈ gPN.A_s) : a_s = (t, p_ac) then
38:               if p_ac ∈ gPN.P_sac then
39:                  outEvRef ← CreateOutEvRef(p_ac)
40:                  t.AddOutEvRef(outEvRef)
41:               end if
42:               if p_ac ∈ gPN.P_nac then
43:                  isNACsource ← TRUE
44:                  outEvRefs.Add(CreateOutEvRef(p_ac))
45:               end if
46:            end if
47:         end for
48:         if isNACsource = TRUE then
49:            AddNewT(tnotenabled, td, inEvRef, outEvRefs)
50:            AddNewPlace(pxor, marking = 1)
51:            AddTwoNewArcs(t, pxor)
52:            AddTwoNewArcs(tnotenabled, pxor)
53:            AddNewPriorityHigherLower(t, tnotenabled)
54:         end if
55:      end if
56:   end for
57:   for all a ∈ nPN.A : a = (x, y) do
58:      if gPN.td(x) ≠ td ∨ gPN.td(y) ≠ td then
59:         nPN.RemoveArc(a)
60:      end if
61:   end for
62:   for all a_s ∈ nPN.A_s do
63:      nPN.RemoveArc(a_s)
64:   end for
65:   for all a_t ∈ nPN.A_t do
66:      nPN.RemoveArc(a_t)
67:   end for
68:   CreateNewPNML(nPN)
69: end for
```

Fig. 3. Decomposition algorithm that reads the global model and creates the components' submodels that support their implementation.

model-checking tool), and implementation [using automatic C and very high speed integrated circuit (VHSIC) hardware description language (VHDL) code generators [43], [44]]. To implement the communication nodes, based on the information

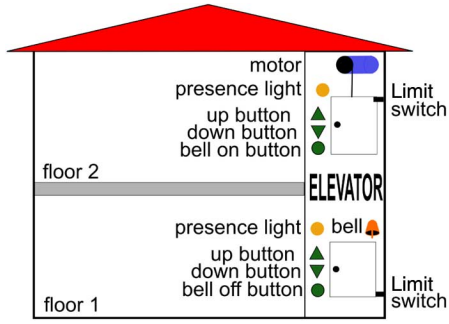

Fig. 4. Small goods lift layout.

about the required resources (obtained in the associated state-space), the approaches proposed in [45] and [46] were used.

To illustrate the application of the defined channels using the extended Petri net class and associated tools, this section presents an application example. The distributed controller of the small goods lift, represented in Fig. 4, was manually specified as presented in Fig. 5. The model specifies a controller with three distributed components (as presented in Fig. 6): component 1 controlling the second floor, with an elevator door, two push buttons (one to go up and the other to go down), and a push button to turn ON a hurry bell on the other floor; component 2 controlling the motor and the limit switches (each floor has a limit switch); and component 3 controlling the first floor, with an elevator door, two push buttons (to go up and to go down), the bell, and a push button to turn OFF the bell. Each door has a lock and a sensor indicating if it is open or closed. In the initial state, the elevator is in the first floor (place *floor1* marked), the elevator is stopped (*motorUp* and *motorDown* unmarked), the first floor door is open (*d1open* marked), the second floor door is locked (*d2locked* marked), no request has been registered (*b2DownOff*, *b2UpOff*, *b1DownOff*, and *b1UpOff* unmarked), and the bell is OFF (*hurryBell* unmarked).

Due to space constraints just a small part of the controller model is described. Each time the button that is in the second floor to go down is pressed (it is associated to transition *b2DownEv*), a message is sent (from the component 1) through the SimpleAC $C1$ to transitions $t22$ and $t12$ (of the component 2). If there is no previous request to go down (*noReqDown* marked) and the elevator is not in floor 1 (*notFloor1* marked), the place *reqDown* is marked. Additionally, if the elevator is not in floor 1, $t12$ fires and a message is sent through the SimpleAC $C3$, otherwise $t12$ does not fire and a message is sent through the NotAC $C2$. When *reqDown* is marked and the elevator is stopped (*stop* marked), place $p9$ is marked, and one message is sent through the SimpleAC $C6$ to lock the floor 2 door. When the door is closed ($d2$_closed marked), $t5$ fires, the door is locked ($d2$_locked is marked), and a message is sent through the SimpleAC $C7$ to fire $t20$, turning ON the motor to go down (*motorDown* is marked). Whenever the turn ON bell is pressed, one message is sent through the SimpleAC $C25$ to turn ON the bell. If the bell is ON ($p15$ marked), $t38$ fires and the bell is turned ON (*hurryBell* is marked), if the bell was already ON, the $t38$ does not fire. In both situations, a message in sent through the AckAC $C26$ to mark place $p6$, and the turn on bell process can be repeated.

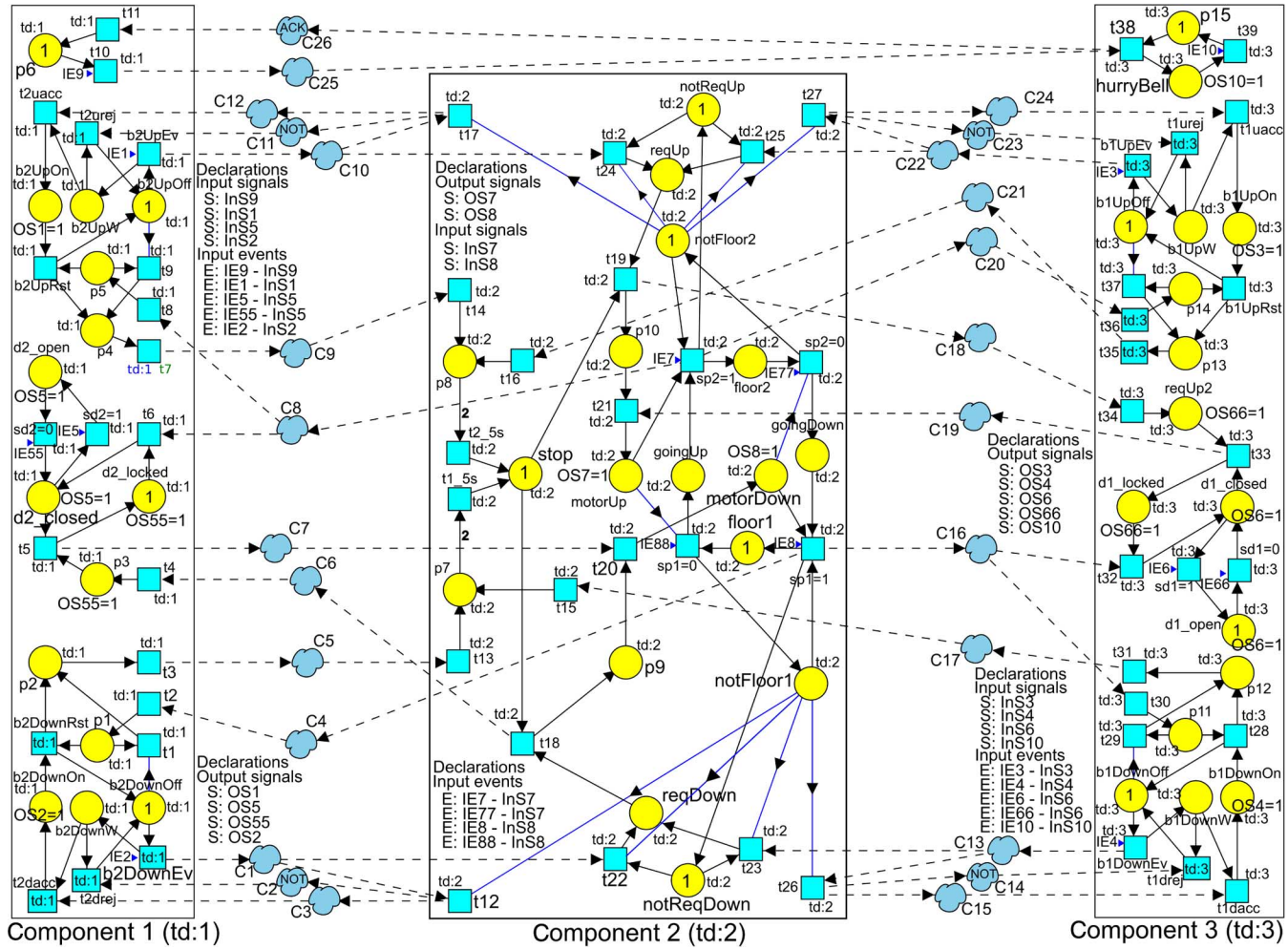

Fig. 5. Distributed controller model of a small goods lift.

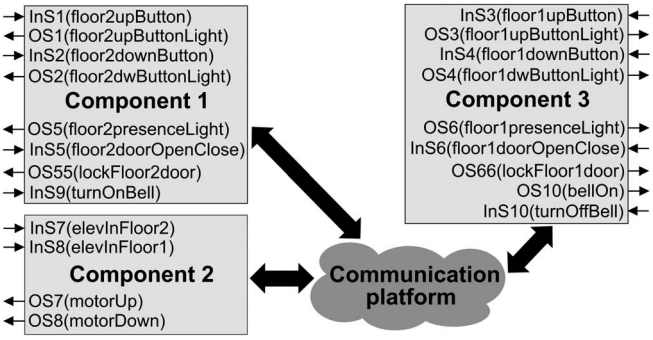

Fig. 6. Small goods lift controller block diagram.

After the system specification, the model was validated using the model-checking tool [25], which generates the associated state-space and relies on a query engine to extract proprieties. It was verified that the system model has a state space with 177 468 states and has no deadlocks. Additionally, a set of significant proprieties were also verified, for instance, that there is no state where the motor is ON and the doors are unlock *(motorDown = 1* OR *motorUp = 1)* AND *(d1_locked = 0* OR *d2_locked = 0)*, as required. This tool also provides the memory resources length, which is required to implement the controllers and the communication channels [45], [46]. After proprieties verification, automatic code generators (such as [43], [44]) can be used to obtain the implementation code to be directly deployed into execution platforms, considering three components/controllers. Besides being platform-independent, the specification is also network-independent, supporting the implementation using different communication channels, network topologies, and network protocols.

VII. Conclusion

Petri net classes extended with the defined ACs and with time domains, priorities, inputs, and outputs, support a model-MBD for DESs composed by synchronous and deterministic components in interaction (GALS-DES). This MBD approach includes specification, simulation (using simulation tools), verification (using model-checking tools), and implementation (using automatic code generators). These Petri net classes ensure that the created models are always GALS, locally deterministic, distributable, network-independent, and platform-independent, and unambiguously identifying the components interaction and components computation, ensuring models readability both by modelers and design automation tools. Additionally, these classes focus the modelers in the components' specification and not on the communication details.

These models provide high flexibility in the implementation phase, supporting the deployment into the most suited implementation platforms, operating at the most suited execution frequency, using the most suited communication channels, network protocols, and network topologies, to interconnect the distributed components. This will simplify the achievement of the desired performance, power consumption, EMI, and production cost, and will also support porting to future implementations in new platforms. These models support the implementation (using automatic code generators) in heterogeneous platforms [such as micro-controllers and field programmable gate arrays (FPGAs)], with safe or unsafe communication channels, with different network protocols [such as Ethernet, controller area network (CAN), and Profibus], and with different network topologies such as point-to-point, star, bus, and ring.

To validate the proposed ACs, the associated MBD approach, and the proposed algorithms, the tool chain framework (http://gres.uninova.pt/) was extended during this work and used to develop distributed GALS systems. The model edition tool was extended to support the creation of models with the proposed channels. The translation algorithm was implemented in the model-checking tool to support GALS-DES models' validation. The extended model-checking tool supports the behavioral verification of the created models (allowing bugs identification and correction during the design phase of the project) and the dimension of the resources that are required to implement the components and the communication nodes. The proposed decomposition algorithm was implemented to support the global model decomposition into a set of implementable submodels, which can be used as input in automatic code (C code and VHDL code) generators. Model-checking tools and automatic code generators can reduce development errors and time. The small goods lift distributed controller model, presented in this paper, illustrates the use of the defined channels, the associated MBD approach, and the extended tool chain framework.

REFERENCES

[1] T. Nolte and R. Passerone, "Guest editorial special section on real-time and (networked) embedded systems," *IEEE Trans. Ind. Informat.*, vol. 5, no. 3, pp. 198–201, Aug. 2009.

[2] Object Management Group. (2013) [Online]. Available: http://www.omg.org/

[3] V. Vyatkin and H.-M. Hanisch, "Bringing the model-based verification of distributed control systems into the engineering practice," in *Proc. 6th IFAC Workshop Intell. Manuf. Syst.*, Poznan, Poland, Apr. 2001, pp. 152–157.

[4] N. Hagge and B. Wagner, "A new function block modeling language based on Petri nets for automatic code generation," *IEEE Trans. Ind. Informat.*, vol. 1, no. 4, pp. 226–237, Nov. 2005.

[5] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli, "Synthesis of multitask implementations of simulink models with minimum delays," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 637–651, Nov. 2010.

[6] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 302–310, May 2012.

[7] S. Sanchez-Solano, M. Brox, E. del Toro, P. Brox, and I. Baturone, "Model-based design methodology for rapid development of fuzzy controllers on FPGAs," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1361–1370, Aug. 2013.

[8] I. Bicchierai, G. Bucci, L. Carnevali, and E. Vicario, "Combining UML-MARTE and preemptive time Petri nets: An industrial case study," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 1806–1818, Nov. 2013.

[9] M. Wehrmeister, C. Pereira, and F. Rammig, "Aspect-oriented model-driven engineering for embedded systems applied to automation systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2373–2386, Nov. 2013.

[10] M. Pajic *et al.*, "Model-driven safety analysis of closed-loop medical systems," *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 3–16, Feb. 2014.

[11] D. Harel, "Biting the silver bullet: Toward a brighter future for system development," *Computer*, vol. 25, pp. 8–20, 1992.

[12] W. Reisig, *Petri Nets: An Introduction*. New York, NY, USA: Springer, 1985.

[13] R. Zurawski and M. Zhou, "Petri nets and industrial applications: A tutorial," *IEEE Trans. Ind. Electron.*, vol. 41, no. 6, pp. 567–583, Dec. 1994.

[14] C. Girault and R. Valk, *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. New York, NY, USA: Springer, 2003.

[15] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves, "A primer on the Petri net markup language and ISO/IEC 15909-2," *Petri Net Newslett.*, vol. 24, no. 76, pp. 9–28, Oct. 2009, presented at the 10th International Workshop Practical Use of Colored Petri Nets and the CPN Tools (CPN'09).

[16] M. Moalla, J. Pulou, and J. Sifakis, "Synchronized Petri nets: A model for the description of non-autonomous sytems," in *Mathematical Foundations of Computer Science*, vol. 64, J. Winkowski, Ed. Berlin, Germany: Springer-Verlag, 1978, pp. 374–384.

[17] M. Rausch and H. M. Hanisch, "Net condition/event systems with multiple condition outputs," in *Proc. 1995 INRIA/IEEE Symp. Emerg. Technol. Factory Autom. (ETFA'95)*, 1995, vol. 1, pp. 592–600.

[18] H.-M. Hanisch and A. Lüder, "A signal extension for Petri nets and its use in controller design," *Fundam. Informat.*, vol. 41, no. 4, pp. 415–431, Dec. 2000.

[19] G. Frey and M. Minas, "Editing, visualizing, and implementing signal interpreted Petri nets," in *Proc. 7th Workshop Algorithmen und Werkzeuge für Petrinetze (AWPN'00)*, Koblenz, Germany, Oct. 2000, pp. 57–62.

[20] L. Gomes, J. Barros, A. Costa, and R. Nunes, "The input–output place-transition Petri net class and associated tools," in *Proc. 5th IEEE Int. Conf. Ind. Informat. (INDIN'07)*, Vienna, Austria, Jul. 2007, pp. 509– 514.

[21] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 1984.

[22] M. E. Grass, F. K. Gürkaynak, and P. Vivet, "Globally asynchronous, locally synchronous circuits: Overview and outlook," *IEEE Des. Test Comput.*, vol. 24, pp. 430–441, Sep. 2007.

[23] F. K. Grkaynak, S. Oetiker, N. Felber, H. Kaeslin, and W. Fichtner, "Is there hope for GALS in the future?" in *Proc. 4th ACiD-WG Workshop Eur. Comm. Fifth Framework Prog.*, Turku, Finland, Jun. 28–29, 2004.

[24] L. H. Yoong, G. Shaw, P. Roop, and Z. Salcic, "Synthesizing globally asynchronous locally synchronous systems with IEC 61499," *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 1465–1477, Nov. 2012.

[25] F. Pereira, F. Moutinho, and L. Gomes, "Model-checking framework for embedded systems controllers development using IOPT Petri nets," in *Proc. IEEE Int. Symp. Ind. Electron. (ISIE)*, May 2012, pp. 1399–1404.

[26] V. Vyatkin and H.-M. Hanisch, "Practice of modeling and verification of distributed controllers using signal net systems," in *Report: Proc. Int. Workshop Concurrency, Specification Prog.*, Oct. 2000, pp. 335–350.

[27] P. Starke and S. Roch, "Analysing signal net systems," Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany, Informatik-Bericht 162, Sep. 2002.

[28] M. Nielsen, V. Sassone, and J. Srba, "Towards a notion of distributed time for Petri nets," in *Applications and Theory of Petri Nets 2001*, J-M. Colom, M. Koutny, Eds. Berlin, Germany: Springer-Verlag, vol. 2075, pp. 23–31, 2001.

[29] H. Kleijn, M. Koutny, and G. Rozenberg, "Processes of Petri nets with localities," School Comput. Sci., Newcastle University Upon Tyne, Tyne, U.K., Tech. Rep. CS-TR-941, Jan. 2006.

[30] M. Koutny and M. Pietkiewicz-Koutny, "Transition systems of elementary net systems with localities," in *CONCUR 2006 Concurrency Theory*, vol. 4137, C. Baier and H. Hermanns, Eds. Berlin, Germany: Springer-Verlag, 2006, pp. 173–187.

[31] F. Moutinho and L. Gomes, "Asynchronous-channels and time-domains extending Petri nets for GALS systems," in *Technological Innovation for Value Creation*, L. Camarinha-Matos, E. Shahamatnia, and G. Nunes, Eds. New York, NY, USA: Springer, 2012, vol. 372, pp. 143–150.

[32] F.-Y. Wang, K. Gildea, H. Jungnitz, and D. Chen, "Protocol design and performance analysis for manufacturing message specification: A Petri net approach," *IEEE Trans. Ind. Electron.*, vol. 41, no. 6, pp. 641–653, Dec. 1994.
[33] J. Billington, S. Vanit-Anunchai, and G. Gallasch, "Parameterised coloured Petri net channel model," in *Transactions on Petri Nets and Other Models of Concurrency*, K. Jensen, J. Billington, and M. Koutny, Eds. Berlin, Germany: Springer-Verlag, 2009, vol. 5800.
[34] S. Christensen and N. Damgaard Hansen, "Coloured Petri nets extended with channels for synchronous communication," in *Application and Theory of Petri Nets*, vol. 815, R. Valette, Ed. Berlin, Germany: Springer-Verlag, 1994, pp. 159–178.
[35] C. Maier and D. Moldt, "Object coloured Petri nets—A formal technique for object oriented modelling," in *Concurrent Object-Oriented Programming and Petri Nets*, G. Agha, F. Cindio, and G. Rozenberg, Eds. Berlin, Germany: Springer-Verlag, 2001, pp. 406–427.
[36] A. Costa and L. Gomes, "Petri net partitioning using net splitting operation," in *Proc. 7th IEEE Int. Conf. Ind. Informat. (INDIN'09)*, Cardiff, U.K., Jun. 2009.
[37] S. Christensen and L. Petrucci, "Modular analysis of Petri nets," *Comput. J.*, vol. 43, no. 3, pp. 224–242, 2000.
[38] G. Liu, C. Jiang, and M. Zhou, "Process nets with channels," *IEEE Trans. Syst. Man Cybern. A, Syst. Humans*, vol. 42, no. 1, pp. 213–225, Jan. 2012.
[39] F. Moutinho and L. Gomes, "Towards distributed execution of Petri net conflicts through model transformation," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Feb. 2013, pp. 1416–1421.
[40] F. Moutinho and L. Gomes, "Augmenting high-level Petri nets to support GALS distributed embedded systems specification," in *Technological Innovation for the Internet of Things*, Camarinha-Matos, Ed. New York, NY, USA: Springer, 2013, pp. 221–228.
[41] R. David and H. Alla, "Non-autonomous Petri nets," in *Discrete, Continuous, and Hybrid Petri Nets*. Berlin, Germany: Springer-Verlag, 2010, pp. 61–116.
[42] J. L. M. Grevet, L. Jandura, J. Brode, and A. H. Levis, "Execution strategies for Petri net simulations," Lab. Inf. Decision Syst., Massachusetts Inst. Tech., Cambridge, MA, USA, LIDS-P-1739 newsletterInfo: 32, 1988.
[43] R. Campos-Rebelo, F. Pereira, F. Moutinho, and L. Gomes, "From IOPT Petri nets to C: An automatic code generator tool," in *Proc. 9th IEEE Int. Conf. Ind. Informat.*, Jul. 2011, pp. 390–395.
[44] F. Pereira and L. Gomes, "Automatic synthesis of VHDL hardware components from IOPT Petri net models," in *Proc. 39th Annu. Conf. IEEE Ind. Electron. Soc. (IECON'13)*, Vienna, Austria, Nov. 2013, pp. 2214–2219.
[45] F. Moutinho, L. Gomes, A. Costa, and J. Pimenta, "Asynchronous wrappers configuration within GALS systems specified by Petri nets," in *Proc. IEEE Int. Symp. Ind. Electron. (ISIE)*, May 2012, pp. 1357–1362.
[46] F. Moutinho, J. Pimenta, and L. Gomes, "Configuring communication nodes for networked embedded systems specified by Petri nets," in *Proc. IEEE Int. Symp. Ind. Electron. (ISIE)*, May 2013, pp. 1–6.

Filipe Moutinho received the Engineering and M.Sc. degrees in electrical and computer engineering from the Faculty of Sciences and Technology (FCT), Universidade Nova de Lisboa (UNL), Lisbon, Portugal, in 2003 and 2009, respectively. Currently, he is pursuing the Ph.D. degree in electrical and computer engineering at the Universidade Nova de Lisboa.

From 2002 to 2006, he was a Junior Teaching Assistant in the area of computational and perceptional systems with the Department of Electrical Engineering, FCT, UNL. From 2006 to 2007, he was a Teaching Assistant at Escola Náutica Infante D. Henrique, Oeiras, Portugal. Until 2009, he was a Software Engineer with the Enterprise NewHotel Software, Lisbon. He worked on several research projects at the Center of Technology and Systems (CTS), UNINOVA Institute, Lisbon. His research interest includes the model-based development of embedded systems.

Luís Gomes (M'96–SM'06) received the Electrotech Engineering degree from the Universidade Técnica de Lisboa, Lisbon, Portugal, in 1981, and the Ph.D. degree in digital systems from the Universidade Nova de Lisboa, Lisbon, in 1997.

He is a Professor with the Electrical Engineering Department, Faculty of Sciences and Technology, Universidade Nova de Lisboa, and a Researcher at the UNINOVA Institute, Lisbon. From 1984 to 1987, he was with Empresa de Investigação e Desenvolvimento de Electrónica SA (EID), Charneca da Caparica, Portugal, a Portuguese medium enterprise. He is an author/coauthor of more than 200 papers published in journals, books, and conference proceedings. His research interests include the usage of Petri nets and other concurrency models applied to reconfigurable and embedded systems codesign.

Dr. Gomes has served as a General Co-Chair or a Program Co-Chair for more than 17 IEEE conferences. He serves as an Associate Editor for the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS (2009–present), and for the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS (2005–2008, 2011–present), among other editorial boards.