2026 IEEE/SICE International Symposium on System Integration (SII)
January 11-14, 2026. Cancún, México

Integrating Heterogeneous Communication Support in Model-Driven Development of Petri nets based Distributed Controllers

Luis Gomes
NOVA University Lisbon
Center of Technology and Systems, UNINOVA
Intelligent Systems Associate LAboratory
Caparica, Portugal
E-mail: lugo@fct.unl.pt

Duarte Tavares
NOVA University Lisbon
NOVA School of Sciences and Technology
Caparica, Portugal
Email: d.tavares@campus.fct.unl.pt

Abstract— **The development of distributed embedded controllers can be significantly enhanced by adopting a model-driven development approach benefiting from the adoption of low-code strategies, where Petri net-based modeling can support rigorous system specification, verification, automatic code generation and direct deployment. This paper presents a development flow where, starting from the global system model expressed through an IOPT Petri net model, a decomposition strategy allow to obtain a set of concurrent sub-models. These sub-models are associated with different time domains in terms of execution time and deployed into a set of networked controllers. Several common protocols can be selected to support inter-controller communication ensuring synchronization between the networked controllers, namely I²C, UART and TCP/MQTT. The model-driven development approach is supported by the IOPT-Tools framework, a web-based platform freely available at `http://gres.uninova.pt/IOPT-Tools/`.**

I. Introduction

Model-based development methodologies have been extensively adopted in various engineering domains. In particular, the model-driven development paradigm (MDD) [1] has emerged as a widely used approach to system design, due to its ability to support automation throughout the development process, facilitated by advanced computational design automation tools. Starting from the system model, designers can rely on simulation environments and specialized analysis frameworks to verify and validate system properties, while being supported by automatic code generation tools. This process enables the direct production of executable code that can be deployed on target implementation platforms with minimal manual programming effort, benefiting from the adoption of low-code development strategies.

Considering the area of distributed embedded controllers, the increasing complexity of systems introduces new challenges in their development. Addressing these challenges often requires starting from the global system level and decomposing the system into distinct components, which may be physically distributed. For that end, specific (potentially heterogeneous) communication support needs to be in place to support inter-controller synchronization.

Among the various modeling formalisms that provide an operational semantics suitable for full support of a model-driven development approach, Petri nets [2] stand out as particularly adequate, combining a graphical representation facilitating communication and interaction among project teams and stakeholders, with rigorous execution semantics and associated textual notation, which fully support seamless integration with computational tools.

Several classes of Petri nets have been introduced, broadly categorized into low-level and high-level nets. Low-level Petri nets include Elementary nets [3] and Place-Transition nets [4], in which tokens are indistinguishable and typically represented either as dots or as numerical annotations indicating their multiplicity. High-level Petri nets, by contrast, encompass formalisms such as Coloured Petri nets [5], where tokens are distinguishable and may carry structured data values.

Within the specific domain of discrete event controller development, several Petri net variants have been proposed [6]. For the specific areas of automation systems and cyber-physical systems, explicit modeling of interactions with the environment is paramount, such as input and output signals and events. For that end, several Petri net classes, normally referred to as non-autonomous Petri nets, have been proposed. Representative examples include synchronized and interpreted Petri nets [7], [8], which incorporate input–output signal dependencies directly into the model execution semantics. Among these non-autonomous Petri net classes, this paper focuses on the Input-Output Place-Transition (IOPT) nets [9] and their supporting toolchain, the IOPT-Tools framework [10].

One of the main objectives of this paper is to present how IOPT nets and the associated IOPT-Tools framework can be jointly employed to support low-code development of distributed controller systems relying on a model-driven engineering paradigm, allowing automatic generation of execution code amenable to be directly deployed into selected implementation platforms. The main contribution is to present how the integration of these distributed controllers can be achieved with the support of associated communication nodes that support heterogeneous protocols and topologies.

The paper is structured as follows. The next section summarizes the main characteristics of IOPT nets, and briefly presents the associated IOPT-Tools framework. Section III

briefly presents the underlying model-driven approach for the development of distributed controllers. Section IV briefly summarizes different common solutions for communication support among heterogeneous implementation platforms and topologies. Section V presents an application example in the automation area using the proposed approach. Finally, Section VI concludes and introduces future work.

II. IOPT NETS AND ASSOCIATED IOPT-TOOLS

This section provides a brief overview of the IOPT net class and its execution semantics. A detailed formal specification of its syntax and semantics can be found elsewhere [9].

IOPT nets extend traditional Place-Transition nets [4] in a non-autonomous manner by introducing input and output signals and events dependencies into the Petri net graph, similarly to approaches from interpreted and synchronized nets [7], [8]. Input signals and events constrain the transition firing, while output signals and events can be produced as a function of the current marking or as a result of transition firings.

IOPT nets are explicitly tailored for the modeling of discrete-event controllers. As common practice in this domain, the model execution adopts a step-based semantics. To ensure a deterministic behavior of the resulting execution, which is of paramount importance in automation controllers applications, a maximal-step execution strategy is adopted. In this sense, all transitions that are enabled and ready are fired simultaneously in the same execution step.

IOPT nets are formally presented in [9], where the autonomous component of the model is extended through the addition of test arcs, while the non-autonomous part also considers transition priorities.

Additional concepts included in the definitions of IOPT nets extend the formalism to address the modeling of Globally Asynchronous Locally Synchronous (GALS) systems [13], where distributed execution is required, as a global clock to execute the system model is no longer possible to consider. To this end, the notions of time domains and communication channels were introduced to capture communication between distinct sub-models, each associated with its own time domain and controller unit. As communication channels exhibit place-like semantics, they connect to transitions through a new type of arc, the communication channel arcs.

In the context of this paper, asynchronous communication channels are considered, allowing asynchronous interaction between sub-models (controllers) associated with different time domains.

The IOPT-Tools framework [10], has been developed considering IOPT nets as the underlying formalism, and is freely accessible at `http://gres.uninova.pt/IOPT-Tools/`. IOPT-Tools contribute to mitigate the scarcity of tools supporting the complete controller development cycle, from specification to execution code generation, facilitating rapid prototyping for different types of platform.

The framework provides a model editor, a simulator for model testing and validating (both a token-player simulation style and a timing-diagram generator can be used), and automatic C and VHDL code generators enabling direct execution of models on platforms such as Arduino, ESP, Raspberry Pi, and other Linux-based devices, as well as in FPGA or reconfigurable logic. A detailed description of these functionalities is presented in [10], including analysis of behavioral properties through the generation of reachability graphs and a dedicated query system.

The editor supports several net operations, namely the net addition and the net splitting operations. While the net addition can be of great help to allow composition of models and their reusability in new projects, the net splitting operation is of particular interest for the present work, as it allows decomposition of the models considering a specific cutting set of nodes, paving the way to identify a set of interconnected sub-models, which in turn will be associated with the networked controllers system.

III. MODEL-DRIVEN DEVELOPMENT APPROACH

Figure 1 summarizes the model-driven development workflow for distributed embedded controllers.

The workflow may start with the composition of the models of parts of the system (using the net addition operation [11]), which is used to produce the model of the entire system. Alternatively, the system model can also be the first step of the development process (depending on the specific characteristics/complexity of the system under development).

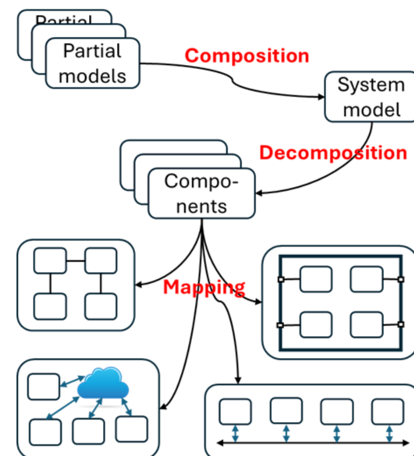

Fig. 1. Underlying development flow.

The decomposition of the system model into a set of concurrent sub-models is obtained through the net splitting operation [12] considering a specific set of cutting nodes. The generated concurrent sub-models obtained through decomposition will have specific time domains associated with them, which will be seen as components ready to be deployable across a variety of implementation platforms, such as networks of distributed industrial controllers (including multi-bus architectures, wired or wireless), System-on-Chip solutions (with or without network-on-chip communication

support), or solutions based on the direct interconnection of generated components, as well as composition of the different referred solutions.

As a result of model decomposition and time domains assignment to the concurrent sub-models, a set of events is generated supporting the communication among the different components and assuring that the behavior of the networked / interconnected controllers is in line with the initial behavior of the whole system.

In this sense, the execution code for each of the controllers will be composed of two parts, as follows:

- The code related with the execution of the IOPT net model of the associated sub-model(s) associated with that time domain (which is provided directly by the IOPT-Tools framework).
- The code related to the communication node, which is responsible for the implementation of the communication events among controllers, considering different protocols and topologies (which currently is manually coded and added, but will be integrated in the IOPT-Tools framework).

IV. Topologies for Communication Support

The decomposition of a global IOPT net model into a set of distributed sub-models needs a robust communication infrastructure to implement the abstract channels defined by the model's cutting set. The selection of an appropriate communication protocol is crucial and depends on the nature of the interaction between sub-models. Adopting a Globally Asynchronous Locally Synchronous (GALS) [13] design paradigm provides a powerful architectural framework for this task. In this framework, each controller sub-model can be realized as a **Locally Synchronous (LS)** island, operating within its own clock domain. Communication *between* these islands is handled asynchronously, forming the **Globally Asynchronous (GA)** system fabric.

A. Locally Synchronous (LS) Communication

For interactions within a single LS island or between tightly-coupled islands (e.g., on the same PCB or SoC), synchronous serial protocols are efficient due to their simplicity and low overhead.

1) I²C (Inter-Integrated Circuit): The I²C protocol is a multi-point serial bus requiring only two lines (SCL and SDA), making it highly suitable for connecting a controller sub-model to multiple local peripheral devices such as sensors or actuators [14]. Although its relatively low speed (up to 3.4 Mbps in high-speed mode) and susceptibility to bus capacitance limit its use for high-throughput data exchange, its simplicity and low pin count make it an excellent choice for control and configuration tasks within a synchronous domain.

2) SPI (Serial Peripheral Interface): Operating in full-duplex mode, the SPI protocol allows simultaneous data transmission and reception over four lines (SCK, MOSI, MISO, SS) [15]. SPI supports significantly higher data rates than I²C, often in the tens of Mbps, making it ideal for implementing high-speed data streams between IOPT net sub-models, such as passing processed sensor data from one module to another. Its primary drawback is the requirement of a dedicated Slave Select (SS) line for each peripheral, which can increase the pin count in complex systems.

B. Globally Asynchronous (GA) Communication

Connecting distinct LS islands, which by definition do not share a common clock, requires asynchronous protocols that can safely manage data transfer across different clock domains. This directly maps to the implementation of token passing between asynchronous sub-nets in the decomposed IOPT model.

1) Asynchronous FIFO with Handshake: For on-chip communication (for example, within an FPGA or SoC), an asynchronous First-In-First-Out (FIFO) buffer with a two-way handshake mechanism is the industry standard solution for reliably crossing clock domains [16]. This method typically employs 'valid' and 'ready' signals to control data flow, ensuring that data are transferred only when the producer has valid data and the consumer is ready to accept them. This perfectly matches the token-passing semantics of Petri nets and is standardized in protocols like the AMBA AXI4-Stream, which uses 'TVALID' and 'TREADY' signals [17].

2) UART (Universal Asynchronous Receiver-Transmitter): For off-chip communication between physically separate components (e.g., two microcontrollers), UART is a classic and robust point-to-point asynchronous protocol [18]. By using a pre-agreed baud rate and framing data with start and stop bits, UART achieves synchronization without a shared clock line. This makes it a straightforward choice for implementing asynchronous channels between IOPT net controllers deployed on separate hardware boards.

C. Network-Level Communication Protocols

When controller sub-models are deployed on nodes distributed across a larger network (e.g., Ethernet or Wi-Fi), IP-based protocols are necessary.

1) TCP (Transmission Control Protocol): TCP provides a reliable, ordered and error-checked data stream over a connection-oriented link [19]. Its robustness makes it suitable for transmitting critical information where data integrity is paramount, such as configuration parameters or high-priority acyclic commands between distributed controllers. However, the overhead associated with its three-way handshake and acknowledgment mechanisms can introduce significant latency, making it less suitable for time-sensitive real-time control loops [20].

2) MQTT (Message Queuing Telemetry Transport): For more complex, event-driven distributed systems, an application layer protocol such as MQTT provides a powerful abstraction over standard transport protocols [21]. MQTT is a lightweight, publish-subscribe messaging protocol designed for constrained devices and low-bandwidth networks. Instead of direct point-to-point communication, components (clients) communicate anonymously through a central message **broker**.

Clients can **publish** messages to specific "topics" or **subscribe** to topics to receive messages. This paradigm is exceptionally well-suited for systems derived from IOPT nets:

- A controller sub-model can publish a message to a topic (e.g., 'conveyor1/status/part-arrived') when a specific transition is fired or a specific place in its net is marked.
- Other sub-models can subscribe to that topic, using the received message as an input event to trigger their own transitions.

This publish-subscribe model decouples the controller sub-models entirely. They only need to be aware of the broker's address and the relevant topics, not the network location or status of every other controller.

V. Models in Action

This section presents an application example from the automation domain to illustrate the efficacy of the proposed methodology. The controller to be developed is designed to control the motors of a three conveyor system, as shown in Figure 2. Each conveyor has two sensors to detect the presence of pieces, one at the entrance and the other at the end of the conveyor (referred to in Figure 2 as in{1...3} and out{1...3}). An additional presence detector (referred to as in4) is activated when a piece is removed from the last conveyor. The foreseen control actions of the controller include activation of the motors of the three conveyors (referred to in Figure 2 as move{1...3}

Fig. 2. Simple conveyor system layout with three conveyors.

A simplified IOPT model allowing the specification of the (centralized) controller is presented in Figure 3. For brevity and clarity, in the model presented, the capacity of each conveyor is of only one piece (however, generalization for any number of pieces is straightforward but will lead to a bigger model, bringing additional readability problems). Also, due to space constraints, only the whole model is presented, even in this system where some symmetries can be identified, and the sub-model of a conveyor could be replicated and further composed using the net addition operation.

Starting with the model in Figure 3, it is possible to identify a cutting set adequate to split the controller into a set of networked controllers, each of them associated with one

Fig. 3. IOPT model of the system's controller.

conveyor. Using the cutting set composed of three transitions and three places, identified as red nodes in Figure 3, four sub-models are obtained. Each sub-model has associated the input sensors (in and out) and motor activation output (move) of the associated conveyor, as well as a set of communication events allowing the synchronization with adjacent controllers. Figure 4 shows the result of the decomposition into a set of concurrent controllers.

Taking into account the execution code for each of the controllers identified on the right-hand side of Figure 4, two components are considered, the first being associated with the execution of the IOPT model itself. The second is associated with the communication support among the controllers and will depend on the specific protocols and topologies selected for the final deployment on the implementation platforms.

Figure 5 shows an example of a possible configuration in which three types of communication support are used:

- An I²C protocol serial bus interconnecting *Controller Two* (master) with *Controller Entrance* and *Controller Three* (slaves).
- MQTT protocol over TCP allowing *Controller Two* to receive events from *Controller Entrance* and *Controller Three*.
- An UART asynchronous serial protocol interconnecting *Controller Three* and *Controller Exit*.

To quantify the resource overhead associated with the proposed heterogeneous communication support, the memory

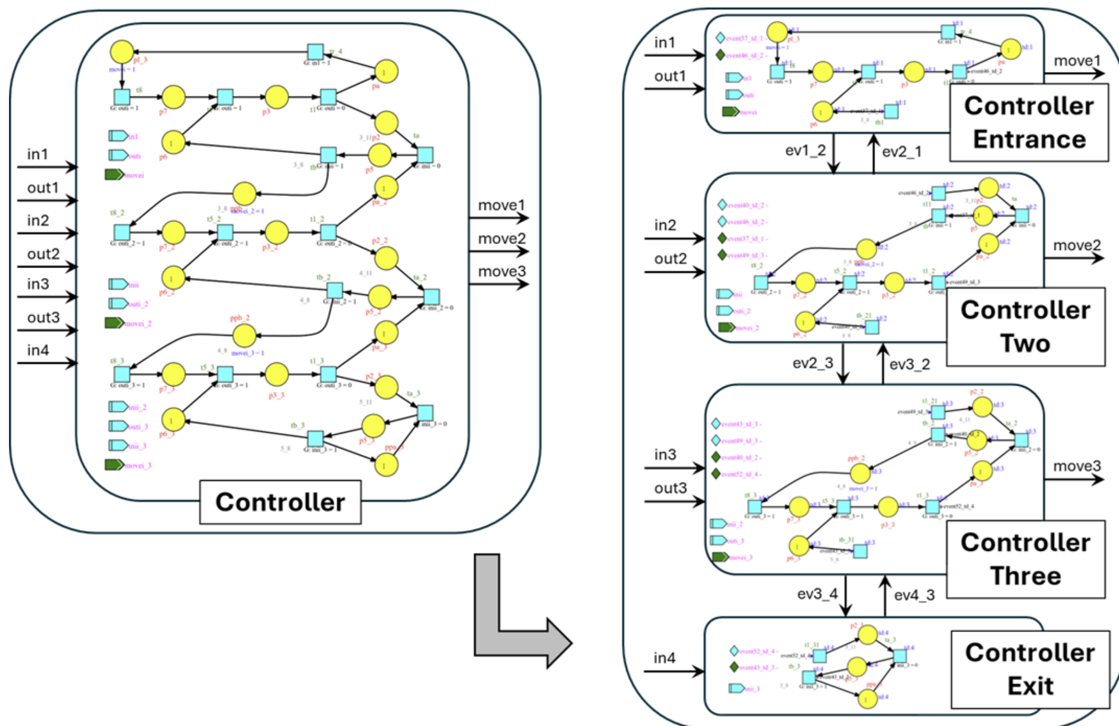

Fig. 4. From centralized controller to a set of networked controllers.

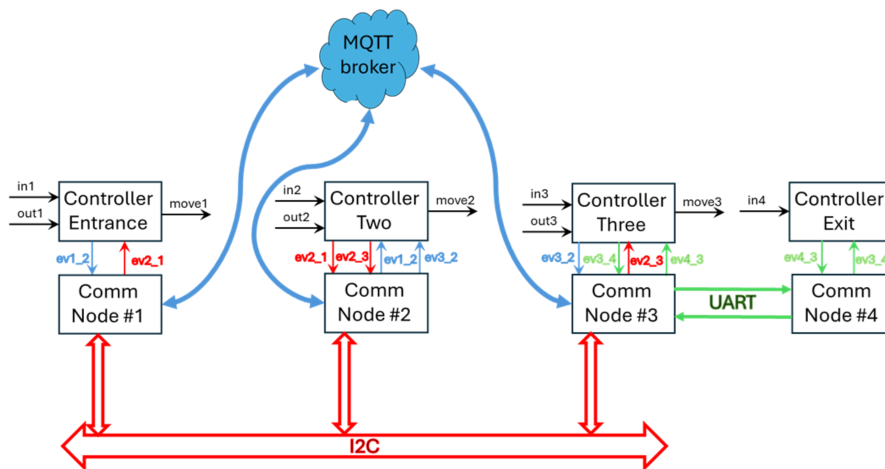

Fig. 5. Possible deployment using different types of protocols for communication/synchronization among networked controllers.

footprint of each controller module was measured with and without its specific communication node present. The analysis presented in Table I reveals a strong correlation between memory overhead and the complexity of the communication protocols implemented.

The Entrance Controller and Controller Two, both implementing a combination of I²C for local communication and a full TCP/IP stack for MQTT messaging, exhibit the most substantial overhead at approximately 26.2 and 27.3 kB, respectively. This shows that network-level protocols are the primary contributors to the memory footprint. The complexity of the TCP/IP stack, which requires significant buffer space and state management for reliable, connection-oriented communication, combined with the MQTT client library, accounts for the vast majority of this resource cost.

TABLE I
MEMORY ANALYSIS OF CONTROLLER IMPLEMENTATIONS.

Controller	Baseline (B)	W/ Comm. (B)	Overhead (B)	Overhead (%)
Entrance	927,970	954,178	26,208	2.82
Two	928,578	955,894	27,316	2.94
Three	928,642	957,014	28,372	3.06
Exit	927,058	928,478	1420	0.15

Controller Three, which includes all the protocols of the first two controllers plus an additional UART interface, shows a slightly higher overhead of 28.4 kB. By comparing

it with the other controllers, we can infer that the marginal cost of adding the UART driver is relatively small, in the range of 1-2 kB.

This hypothesis is decisively confirmed by the Exit Controller. This module, which implements *only* the UART protocol for simple, point-to-point asynchronous communication, has a remarkably low overhead of just 1,420 bytes (a 0.15% increase). This stark contrast highlights the lightweight nature of hardware-level serial protocols compared to their network counterparts.

The protocol latency reported in Table II is the average of 20 measurements obtained from the prototype. The micros() Arduino function was used. For hardware protocols (I²C and UART), the one-way transfer time was calculated by introducing the sender's start timestamp within the message payload and subtracting it from the receiver's end timestamp upon arrival. Conversely, for network-based protocols (TCP and MQTT) utilizing a public broker, the latency was estimated as half the measured Round-Trip Time (RTT), determined via a stable "ping-pong" test configuration.

TABLE II
AVERAGE COMMUNICATION LATENCY MEASUREMENTS.

Protocol	Average Latency	Notes
I^2C (at 400kHz)	185 µs	One-way, master-to-slave transfer.
UART (at 115200 bps)	250 µs	One-way, point-to-point transfer.
TCP/MQTT	45 ms	Estimated one-way latency via public broker.

These findings have significant implications for the design of distributed systems on resource-constrained platforms. The choice of communication protocol represents a critical trade-off between functional flexibility and resource consumption. The model-driven development approach presented in this paper allows a designer to make informed decisions during the partitioning phase of the system. By analyzing the communication requirements of each sub-model, complex and memory-intensive protocols like TCP/MQTT can be selectively deployed only on controllers that require them. Simpler nodes, like the Exit Controller, can be implemented with lightweight protocols, thereby optimizing the overall system's memory usage and ensuring hardware requirements are met.

VI. Conclusion and Future Work

This paper presents a model-driven methodology for the development of distributed controller systems allowing rapid prototyping and adopting low-code development strategies. The approach relies on a specific class of Petri nets, the Input-Output Place-Transition (IOPT) nets, to model system behavior, and in the associated IOPT-Tools framework, a freely accessible set of web-based computational tools available at `http://gres.uninova.pt/IOPT-Tools/`, which support the whole phases of development, including automatic generation of execution code. As future work, the referred communication protocols allowing inter-controller communications will be integrated in the framework ensuring synchronization between the networked controllers.

Acknowledgment

This work was financed by the Portuguese Agency FCT – Fundação para a Ciência e Tecnologia, in the framework of project CTS/00066.

References

[1] B. Selic (2003). The pragmatics of model-driven development. IEEE Software 20 (5), pages 19-25

[2] W. Reisig (1985). Petri nets: an Introduction. Springer-Verlag New York, Inc. ISBN 0-387-13723-8

[3] G. Rozenberg and J. Engelfriet (1998). Elementary net systems. In: Lectures on Petri Nets I: Basic Models: Advances in Petri Nets. W. Reisig, G. Rozenberg (Editors). Springer Berlin Heidelberg. pp. 12–121. url=https://doi.org/10.1007/3-540-65306-6_14

[4] J. Desel and W. Reisig (1998). Place/transition Petri Nets. In: Lectures on Petri Nets I: Basic Models: Advances in Petri Nets. W. Reisig, G. Rozenberg (Editors). Springer Berlin Heidelberg. pp. 122–173. url=https://doi.org/10.1007/3-540-65306-6_15

[5] K. Jensen and L. Kristensen (2009). Coloured Petri Nets: Modelling and Validation of Concurrent Systems. ISBN: 978-3-642-00283-0. DOI: 10.1007/b95112

[6] C. Girault and R. Valk (2001). Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. ISBN: 3540412174. Springer-Verlag

[7] M. Silva (1985). Las Redes de Petri: en la Automática y la Informática, Editorial AC Madrid

[8] R. David and H. Alla (2010). Discrete, Continuous, and Hybrid Petri Nets. ISBN: 3642106684. Springer Publishing Company, Incorporated

[9] L. Gomes and J. P. Barros (2018). Refining IOPT Petri Nets Class for Embedded System Controller Modeling. IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society. DOI 10.1109/IECON.2018.8592921

[10] F. Pereira, F. Moutinho, A. Costa, J. P. Barros, R. Campos-Rebelo and L. Gomes (2022). IOPT-Tools – From executable models to automatic code generation for embedded controllers development. PETRI NETS 2022 - 43rd Int. Conf. on Application and Theory of Petri Nets and Concurrency; Bergen, Norway; LNCS 13288, pp. 127–138, 2022. https://doi.org/10.1007/978-3-031-06653-5_7

[11] J.-P. Barros and L. Gomes (2004). Net Model Composition and Modification by Net Operations: a Pragmatic Approach. in INDIN'2004 – 2nd IEEE International Conference on Industrial Informatics; 24-26 June 2004; Berlin, Germany. DOI: 10.1109/INDIN.2004.1417350

[12] A. Costa and L. Gomes (2007). Petri net Splitting Operation within Embedded Systems Co-design. in 2007 5th IEEE International Conference on Industrial Informatics, 23-26 July 2007, Vienna, Austria, DOI: 10.1109/INDIN.2007.4384808.

[13] F. Moutinho and L. Gomes (2015). Asynchronous-channels within Petri net based GALS distributed embedded systems modeling. IEEE Transactions on Industrial Informatics; Vol.10 Issue: 4; Nov. 2014; pp 2024-2033; ISSN: 1551-3203; DOI: 10.1109/TII.2014.2341933

[14] NXP Semiconductors, "I²C-bus specification and user manual," Rev. 6, UM10204, April 2014.

[15] J. F. Wakerly, *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2006.

[16] L. Bening and H. D. Foster, *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*, 2nd ed. Boston, MA: Springer Science+Business Media, 2002.

[17] ARM, "AMBA 4 AXI4-Stream Protocol Specification," Protocol Specification IHI 0051A, March 2010.

[18] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Hoboken, NJ: Pearson, 2021.

[19] J. Postel, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, Sep. 1981. [Online]. Available: `https://www.ietf.org/rfc/rfc793.txt`

[20] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley Professional, 1994.

[21] A. Banks and R. Gupta, Eds., "MQTT Version 3.1.1," OASIS Standard, Oct. 2014. [Online]. Available: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`