NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

DUARTE JOSÉ RIBEIRO TAVARES

BSc in Electrical and Computer Engineering

COMMUNICATION MODULES FOR DISTRIBUTED CONTROLLERS SPECIFIED THROUGH IOPT MODELS

Dissertation Plan
MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
July, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

COMMUNICATION MODULES FOR DISTRIBUTED CONTROLLERS SPECIFIED THROUGH IOPT MODELS

DUARTE JOSÉ RIBEIRO TAVARES

BSc in Electrical and Computer Engineering

Adviser: Luís Filipe dos Santos Gomes
Associate Professor with Habilitation, NOVA University Lisbon

Dissertation Plan
MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
July, 2025

Contents

List of Figures

1 Introduction

In this chapter, the context of the dissertation plan is established with respect to previously developed related work, thereby outlining the motivation for the topic and the intended objectives. A concise overview of the dissertation's structure is also provided.

1.1 Context and Motivation of the Work

The concept of Petri nets was first introduced by Carl Adam Petri in his 1962 dissertation, "Kommunikation mit Automaten" [25]. Originally proposed as a modeling technique for distributed and concurrent systems in computer science, Petri nets have since become widely adopted in the fields of embedded systems and software development.

Petri nets provide a robust and intuitive framework for the modeling and analysis of complex systems. Their graphical representation facilitates the visualization of concurrent, asynchronous, and distributed processes, while their underlying mathematical formalism enables rigorous validation and verification. Due to their versatility and expressive power, Petri nets have proven to be a valuable tool for both theoretical research and practical engineering applications [22].

Among the numerous tools available for Petri net modeling, the IOPT-Tools environment underpins a specialized approach to developing controllers through an IOPT models, a specific class of Petri nets. The IOPT-Tools framework, accessible at `http://gres.uninova.pt/IOPTTools/`, provides mechanisms for decomposing complex models into independent sub-models, allowing them to execute across distinct computational nodes.

The increasing complexity of distributed control system models, particularly under the GALS (Globally Asynchronous, Locally Synchronous) paradigm, has intensified the need for efficient and dependable communication between these distributed sub-models. As systems become more modular and geographically distributed, guaranteeing low-latency data exchange and maintaining operational consistency pose significant challenges. These challenges are particularly acute in the context of distributed IOPT sub-models and GALS, as manual communication handling can lead to extensive development effort, introduce

potential for errors, and complicate the formal verification of system behavior.

1.2 Problem Statement

At the heart of this work lies the challenge of integrating effective communication channels into the IOPT-Tools environment. While IOPT-Tools excels at defining individual model logic and decomposing complex systems, its current capabilities do not extend to the automated generation of model communication infrastructure. The main problem is to establish a robust mechanism that enables the distributed controllers, each executing an independent IOPT sub-model, to exchange data efficiently. This is complicated by the variety of communication technologies available, each with its distinct advantages and constraints, which raises the question of how best to manage this diversity and meet the rigorous performance requirements of distributed automated systems.

1.3 Objectives

This project is organized around two principal objectives:

- Comparative Analysis of Communication Technologies: Conduct a thorough study of both wired point-to-point networks, such as I2C, SPI, and UART and wireless solutions like Wi-Fi and Bluetooth, evaluating them based on criteria such as simplicity, reliability, and latency.
- Development of an Automated Code Generation Tool: Design and implement an algorithm or workflow that can automatically generate the code required for each IOPT sub-model. This mechanism should streamline the setup of efficient data exchanges among controllers, ensuring that the overall system remains robust and responsive to the demands of distributed control.

It is expected that the study of these two objectives will yield an automatic code generation tool capable of producing C or VHDL code, depending on the inputs provided. This tool will be implemented within the IOPT-tools, thereby ensuring reliable and effective communication between IOPT submodels.

1.4 Dissertation Plan Structure

This dissertation plan is organized into four main chapters, each addressing a key component of the research:

- **Chapter 1 – Introduction:** This chapter introduces the context and motivation behind the research, defines the problem statement, and presents the main objectives of the dissertation. Additionally, it outlines the structure of the document, providing a clear guide for the reader.

- **Chapter 2 – State of the Art:** This chapter provides a thorough review of the theoretical and technological foundations that underpin the research. It begins with an introduction to the fundamental concepts of Petri nets, including a discussion of their historical development. The chapter also explores the distinctive characteristics of IOPT Petri nets, with mention of the operations of net addition and subtraction. It further examines Globally Asynchronous Locally Synchronous (GALS) systems and delves into different communication technologies within distributed environments. Finally, the chapter discusses the IOPT-Tools environment and its associated features that supports the research.
- **Chapter 3 – Work Plan:** This chapter describes the planned research methodology, including the characterization of the proposed solution, the construction of the Gantt diagram for project scheduling, and the organization of tasks.
- **Chapter 4 – Conclusion:** The final chapter presents the conclusions drawn from the conducted research, summarizing the contributions and suggesting directions for future work.

2

State of the Art

This chapter establishes the theoretical framework for the development of a distributed controller communication tool. It begins with an exploration of Petri nets, which are foundational in modeling the interactions and behavior of distributed systems. Following this, the chapter discusses GALS (Globally Asynchronous, Locally Synchronous) communication technologies, highlighting their significance in enabling efficient and reliable communication within distributed systems. The discussion then examines various communication protocols, including I2C, SPI, UART, RS-485 and FIFO with handshake, evaluating their suitability and ensuring effective communication within the distributed IOPT controller system .Finally, the chapter introduces the IOPT-Tools environment, focusing on its role in supporting the design and development of the distributed controller communication tool.

2.1 Petri Nets

2.1.1 History

The German computer scientist Carl Adam Petri formalized the concept of Petri nets, as a formalism for modeling distributed and concurrent systems, in his 1962 PhD dissertation, *Kommunikation mit Automaten*, at the Technical University of Darmstadt [25], in the following years, the theoretical foundations of Petri nets were strengthened by seminal results on decision problems such as reachability and liveness [22], while the annual International Conference on Applications and Theory of Petri Nets and Concurrency provided a forum to advance both theory and practice [9], firmly establishing Petri nets as a formal graphical language for discrete-event and concurrent systems modeling [37].

In subsequent decades, the theoretical foundations of Petri nets were strengthened by seminal results on decision problems such as reachability and liveness [22], while the establishment of the International Conference on Applications and Theory of Petri Nets and Concurrency created an annual forum to advance both theory and practice [9], firmly establishing Petri nets as a formal graphical language for discrete-event and concurrent systems modeling [37].

2.1.2 Definition

Petri nets, over the years, have been developed and adapted to better suit different applications, Colored Petri Nets (CPNs), Timed Petri Nets, Hierarchical Petri Nets, Input-Output Place-Transition(IOPT) among others were introduced to better meet these needs.

To manage this diversity, the term **Place/Transition net (P/T net)** was established as the standard name for the classical formalism, a standardization heavily influenced by seminal works such as Murata's [22]. This P/T net serves as the foundational model from which the advanced types mentioned above are derived from. Essentially, every advanced Petri net is an extension of the classical P/T net, inheriting its fundamental concepts of places, transitions, and firing rules. A firm grasp of the P/T net definition is therefore a prerequisite for understanding its more complex variants.

Place/Transition nets are a bipartite directed-graph formalism comprising of primitive elements, *places* (depicted as circles), *transitions* (depicted as bars) and *Arcs* (depicted as arrows), and *tokens* that reside in places to represent system state (see Figure 2.1). The minimality of these primitives enables the construction of richer constructs (e.g., forks, joins) while preserving analytical tractability [31]. Semantically, places model local conditions or resources and transitions denote events whose firing consumes and produces tokens, thereby capturing concurrency, synchronization, conflict and choice within a unified mathematical framework [31].

Graphically, P/T nets serve as intuitive visual-communication aids for stakeholders, such as clients, manufacturers and users, supporting model comprehension and system specification [30].

Mathematically, they admit formalisms like state equations and algebraic invariants for rigorous analysis; however, there is a critical tradeoff between modeling generality and analysis capability, often necessitating application-specific restrictions or tool support for simulation and verification [22].This inherent complexity, where Petri-net-based models can become too large for analysis even for a modest-size system, is mitigated in the IOPT-Tools environment through its support for hierarchical modeling and decomposition into independent sub-models, which aids in managing the complexity for distributed systems.

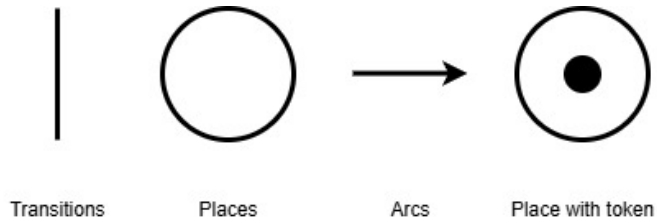

Figure 2.1: Petri net elements.

The formal definition of a P/T net states that a Petri net PN is defined as the tuple (meaning they cannot be modified once created) $PN = (P, T, F, W, M_0)$ [22], where:

- $P = \{p_1, p_2, p_3, \ldots, p_m\}$ is the set of m places;

- $T = \{t_1, t_2, t_3, \ldots, t_n\}$ is the set of n transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs representing the flow relation;
- $W : F \rightarrow \{1, 2, 3, \ldots\}$ is the arc weight function;
- $M_0 : P \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A Petri net structure $N = (P, T, F, W)$ without an initial marking is denoted by N and a Petri net with an initial marking is denoted by (N, M_0).

Figure 2.2 presents a Place/Transition net that models the execution and synchronization of two parallel processes. The model demonstrates a classic fork-join structure, where concurrent tasks are initiated by transition T_0 and must both be completed before synchronizing at transition T_3 to continue the cycle.

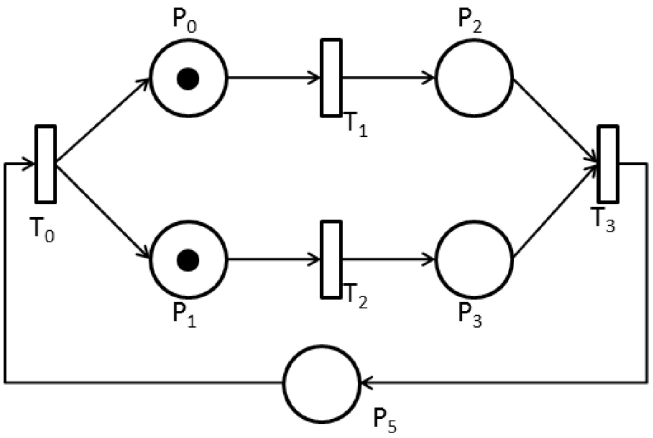

Figure 2.2: Example of Place/Transition net.

2.2 *Input-Output Place-Transition* Petri Nets

The **Input-Output Place-Transition (IOPT) Petri Nets** were created to model controllers interacting with external environments through the use of input and output (I/O) interfaces[14]. They are considered *non-autonomous* where non-autonomous means that external signals can enable or disable transitions [24]. The incorporation of inputs and output signals in Petri nets enables precise representation of the interactions between a controller and its external environment, therefore enabling its applicability in real-world environments and making them particularly useful in automation and embedded systems[14].

The IOPT framework can be defined as a tuple of input signals (IS), input events (IE), output signals (OS), and output events (OE). This design ensures the synchronization of the modeled control logic with the external environment [14]. The introduction of priority

attributes for transitions and the inclusion of test arcs represent notable advancements over earlier versions of IOPT nets, such as those described in previous works in [2] and [23]. The introduction of prioritization allows for effective conflict resolution among transitions, while test arcs facilitate the implementation of fair arbitration mechanisms [10].

Furthermore, the IOPT framework incorporates features such as time domains and communication channels, which support the modeling of networked controllers and globally-asynchronous locally-synchronous (GALS) systems. Its metamodel complies with the Petri Net Markup Language (PNML) and extends it with Ecore-based representations to capture I/O, timing, and communication aspects [14].

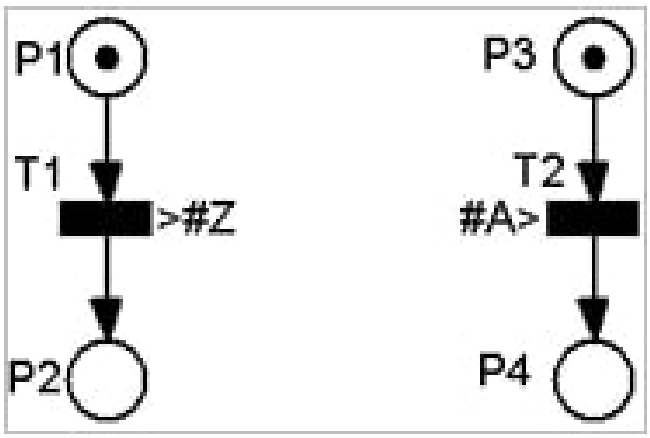

Figure 2.3: Example of IOPT Petri net.

Figure 2.3 presents two Petri nets where transitions are made with communication actions (>#Z on T_1 and #A> on T_2). The figure demonstrates how locally asynchronous components achieve global synchrony. Structurally, the two sub-nets are independent, as the enabling of T_1 and T_2 depends only on their local markings but a transition can only fire if a complementary input/output action occurs simultaneously in its environment. In the state shown, although both transitions are locally enabled, they cannot synchronize with each other as their communication actions do not match; each must await a corresponding partner.

2.2.1 IOPT Net Composition and Decomposition

Net Composition and Decomposition are fundamental operations in the IOPT Petri nets framework, they form the foundation for its adaptability in embedded and distributed systems.The combination and decomposition of two IOPT nets through synchronized transitions and shared places preserve the input/output (I/O) signal dependencies [8].

- **Net Composition:** As the term suggests, net composition functions as a composition operator that integrates two IOPT net modules into a larger model, thereby facilitating the reuse of pre-validated components. Comparable to additive compositionality [33], this operation transforms submodels into coherent systems while

preserving semantic consistency through synchronized transitions and shared places, which maintain the dependencies of input/output (I/O) signals [8].

- **Net Decomposition (or Net Splitting):**

  Net Decomposition, or net splitting, is a formal operation designed to divide Petri net models into a set of smaller, concurrent sub-models.The primary goal is to transform a centralized system specification into distributed components that can be independently implemented on various platforms, such as separate hardware controllers or software processes [6]. The use of net splitting creates smaller and more manageable systems, supporting modular design approaches and allowing for incremental development and analysis [13].

  In net splitting operations, it is essential to identify and validate the nodes, known as the *cutting set*, where the model should be broken. The validity of a cutting set is determined by adhering to the following rules introduced in [5]:

  1. **Rule 1** - Splitting at a Place: This rule is invoked when the cutting node is a place. The fundamental precondition is that after the conceptual removal of the place, its pre-set (input transitions) and post-set (output transitions) are separated into different components.
  2. **Rule 2** - Splitting at a Transition with Single-Component Input: If a transition is the cutting node and all its input places belong to what will become a single component, the transition is kept in that "master" component. A copy of the transition is then placed in the component(s) containing the output places.
  3. **Rule 3** - Splitting at a Transition with Multi-Component Input: If a cutting transition has input places that will belong to different components after the split, one component is designated as the "master." This master component receives the original transition and copies of the input places (and their preceding transitions) from the other components. The other components receive a copy of the cutting transition.

  To maintain behavioral consistency immediately after the split, the operation relies on synchronous communication channels [5], [6]. When a node is split, the original and its copies are linked in a "synchrony set" or "synchrony group" [6]. One transition in the set is designated as the "master" and the others as "slaves" [5], [6]. Semantically, all transitions in a synchrony set are intended to fire simultaneously, as if they were a single fused transition [5]. This ensures that, at the model level, the set of sub-models is behaviorally equivalent or similar to the original, monolithic net [5], [6].

2.3 GALS

The increasing complexity of embedded and distributed systems necessitates modeling approaches that can effectively handle concurrency, modularity, and varying timing constraints. GALS architectures address these needs by allowing system components to operate synchronously within themselves while communicating asynchronously with other components. When integrated into the IOPT Petri net framework, GALS architectures enable a structured and scalable approach to system design and verification.

In GALS (Globally Asynchronous, Locally Synchronous) systems, each local component operates synchronously with respect to its own local clock, which governs its state evolution. However, since each component resides in a distinct clock domain, the overall system exhibits asynchronous behavior. Inter-component communication can be facilitated through asynchronous wrappers, such as those proposed in [4].

An extension of GALS systems applied to IOPT models is presented in [21], where the author introduces an attribute that specifies the Time Domain (TD) of each node within the IOPT Petri net, including both places and transitions. This attribute enables the association of each node with a specific hardware or logical component, thereby facilitating modular and time-partitioned system design. To support communication between components operating in different time domains, the model incorporates the concept of Asynchronous Channels (ACs). An AC connects two transitions that belong to distinct time domains, for example, one acting as the master, responsible for sending events, and the other as the slave, which receives them. These events are transmitted across the AC, enabling coordinated yet asynchronous interaction between independently clocked components.

An IOPT Petri net extended with Time Domains and Asynchronous Channels can be formally defined as follows:

$$\text{IOPT\_GALS} = (\text{IOPT}, \text{ACs}, \text{TDs}) \tag{2.1}$$

where:

1. **IOPT** denotes an IOPT Petri net, defined as in [14];
2. **ACs** is the set of asynchronous channels;
3. **TDs** is the set of time domains.

An IOPT net is formally defined as:

$$\text{IOPT} = (P, T, A, TA, M, weight_T, weight_P, priority, isg, ie, oe, osc) \tag{2.2}$$

The following constraints further define the IOPT-GALS structure:

$$ACs \subseteq T \times T \tag{2.3}$$

$$t_s \times t_m \subseteq ACs \tag{2.4}$$

$$TDs = TDs_p \cup TDs_t \cup TDs_{ac} \tag{2.5}$$

where t_m and t_s denote the master and slave transitions, respectively, such that:

$$t_m \in T, \quad t_s \in T, \quad t_m \neq t_s$$

The mapping functions are defined as:

$$TDs_p : P \rightarrow \text{IN}, \quad TDs_t : T \rightarrow \text{IN}, \quad TDs_{ac} : ACs \rightarrow \text{IN}$$

The integration of GALS into IOPT Petri nets involves decomposing a global system model into locally synchronous modules that communicate asynchronously. This decomposition creates TD and AC, it facilitates modular design, enabling each module to operate at its own pace without the need for global synchronization, thus improving scalability and flexibility [21].

Figure 2.4 illustrates an example of a GALS system. The model depicts two distinct time domains: one on the left side (TD_1) and another on the right side (TD_2). Positioned between them is an asynchronous channel (AC), which facilitates communication across time domains.

All elements on the left side operate synchronously within TD_1, while those on the right side operate synchronously within TD_2. Components sharing the same time domain communicate synchronously. However, communication between components in TD_1 and TD_2 is asynchronous and is mediated by the AC, which is associated with its own independent time domain.

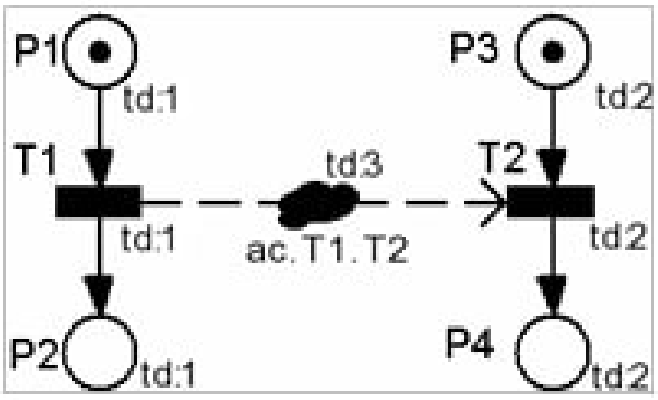

Figure 2.4: Example of Petri net with GALS system.

2.4 Communication support technologies

In this section, various communication technologies utilised within Globally Asynchronous Locally Synchronous (GALS) systems are examined. The analysis focuses on their design considerations, performance trade-offs, and integration challenges. By providing a comprehensive overview of these mechanisms, the chapter aims to equip readers with the knowledge necessary to navigate the complexities of modern integrated circuit development and to harness the benefits of GALS architectures for optimised system performance and reliability.

2.4.1 I^2C

The I^2C (Inter-Integrated Circuit) protocol is described as a multi-master, serial and single-ended bus that requires only two lines for communication: the clock line (SCL) for synchronization and the data line (SDA) for transmitting information. This simplicity is underscored by the minimal hardware requirements, just three connections (SCL, SDA, and GND), making it an ideal choice for connecting the central controller (master) with multiple local controllers (slaves) [26].

Despite its advantages, the I^2C protocol presents certain limitations that may hinder its suitability for high-performance or large-scale applications. The standard I^2C bus supports data rates of up to 3.4 Mbps in high-speed mode, which may prove inadequate for systems that demand greater throughput. Moreover, the inherent bus capacitance imposes constraints on the number of connected devices, the permissible bus length and may have problems with bus contention, thereby limiting the scalability of the system [36].

Since I^2C assumes that both devices operate within the same clock domain, it is not suitable for the asynchronous components of GALS (Globally Asynchronous, Locally Synchronous) systems. However, its synchronous nature makes it a viable option for communication within the locally synchronous parts of IOPT sub-models, where tight timing and minimal wiring are priorities. This would necessitate careful design to ensure proper clock domain crossing if data needs to be exchanged with other asynchronous GALS domains.

2.4.2 SPI

The Serial Peripheral Interface (SPI) protocol operates in Full-Duplex mode, allowing simultaneous data transmission and reception this aligns well with the requirements of GALS architectures [17]. Same as other serial protocols it has a master-slave configuration with a central module (master) to coordinate communication, ensuring data integrity across asynchronous boundaries. It uses four primary lines for communication. These signals include SCK (Serial Clock), MOSI (Master Output Slave Input), MISO (Master Input Slave Output), and SS (Slave Select). The protocol's simplicity and high-speed data

transfer rates, often ranging from 10 Mbps to several tens of Mbps, make it suitable for applications demanding rapid and reliable data exchange adding [17].

The protocol's inherent simplicity, flexibility, and capacity for high-speed data transfer, often ranging from 10 Mbps to several tens of Mbps, render it well-suited for applications requiring rapid and reliable data exchange. Additionally, its ability to simultaneously transmit and receive data, coupled with support for multiple slave devices through individual Slave Select (SS) lines, facilitates scalable system designs.

While the Serial Peripheral Interface (SPI) protocol offers several advantages, it also presents notable limitations. Each additional slave device requires a dedicated Slave Select (SS) line, which can increase the pin count on the master device and complicate hardware design, particularly in systems with numerous peripherals. Moreover, unlike protocols such as I^2C, SPI does not include inherent error-checking mechanisms, necessitating the implementation of additional measures to ensure data integrity. Furthermore, SPI is typically optimized for short-distance communication within a single printed circuit board (PCB), as signal degradation and timing issues may arise over extended distances. These factors can limit the scalability and robustness of SPI in more complex or distributed system architectures [19].

SPI, like I^2C, is a synchronous communication protocol, meaning that data transmission is synchronized by a clock signal generated by the master device. Although this ensures reliable data transfer, SPI is only suitable for the synchronous parts within a GALS (Globally Asynchronous, Locally Synchronous) subsystem and is not appropriate for communication between asynchronous GALS domains. Its high speed makes it attractive for critical data paths within a locally synchronous IOPT sub-model, but integrating it into a globally asynchronous system would require dedicated asynchronous mechanisms at the clock domain boundaries.

2.4.3 UART

The Universal Asynchronous Receiver-Transmitter, UART, is a fundamental component in serial communication systems, particularly in embedded and microcontroller-based applications [38]. It is a hardware asynchronous communication with full-duplex data exchange using two or four signal lines. In its two-signal configuration, communication is carried out via the transmit (TX) and receive (RX) lines. In the four-signal variant, additional control signals, ready-to-send (RTS) and clear-to-send (CTS), are included to enable hardware-based handshaking for improved flow control [29].

UART operates by converting parallel data into serial form for transmission, and performing the reverse operation during reception. The data frame typically includes a start bit (logical low), a defined number of data bits, an optional parity bit for error detection, and one or more stop bits. While UART handles data framing and signal generation, it does not define a standardized signaling protocol between devices, requiring both ends to be properly configured. UART signals are output at the operating voltage of the device,

making them suitable for short-range communication between components operating at identical voltage levels. However, in many practical applications, this condition is not met, for this reason UART signals are often routed through line drivers to convert them into standard electrical signaling formats, such as RS-485, to support longer distances and improve noise immunity [29].

UART communication does not rely on a shared clock signal, instead communicating devices use predefined baud rates to determine the timing of data bits to ensure synchronization [16]. The integration of UART modules within IOPT Petri net models is crucial for the accurate representation and verification of asynchronous communication in GALS systems, as its inherent asynchronous nature aligns directly with the inter-domain communication requirements. This makes UART a strong candidate for facilitating the communication channels between distributed IOPT sub-models.

2.4.4 FIFO + Handshake

FIFO, short for First-In-First-Out, is a protocol that utilizes buffers together with handshake signals. It is a memory buffer that stores data elements in the order of their arrival and retrieves them in the same sequence, thereby adhering to the 'first-in, first-out' principle [35]. To manage and facilitate the data flow of this buffer effectively, a handshake is used, which prevents issues such as corruption or data loss. This communication involves three primary signals: the data bus, which carries the actual information; a valid signal, asserted by the sender or source interface to indicate that the data is stable and available for transfer; and a ready signal, asserted by the receiver or destination interface to signify its preparedness to accept new data [1]. The use of these signals ensures that no data is lost or overwritten and allows for an asynchronous communication system. Such a mechanism is highly efficient, fully decouples the sender and receiver, and is particularly well-suited for implementations in Field-Programmable Gate Arrays (FPGAs) or hardware-in-the-loop systems, often for crossing clock domains [3]. This makes FIFO + Handshake a fundamental building block for robust asynchronous communication between distributed IOPT sub-models in a GALS system, directly addressing the challenge of reliable data exchange across distinct time domains.

2.4.5 IP

IP, the Internet Protocol is a standard for directing and labeling data packets. These standards enable the packets to navigate various networks and reach their intended recipient accurately. These packets are essentially data traversing the Internet that was divided into smaller pieces. IP information is attached to each packet, and this information helps routers to send packets to the right place. An IP address is allocated to each device or domain linked to the Internet. This address directs packets, ensuring data reaches its intended destination [34].

At the destination, received packets undergo processing according to the specific transport protocol integrated with IP. The most frequently employed transport protocols include TCP and UDP, each dictating unique handling procedures.

1. **TCP** Transmission Control Protocol is a connection-oriented protocol designed to provide a reliable, ordered, and error-checked stream of data between applications, a definition established by its original specification [27] and taught widely in foundational networking texts [20]. It establishes a connection via a three-way handshake before data transmission begins, a process detailed exhaustively in technical literature [32]. To ensure reliability, TCP uses mechanisms such as sequence numbers, acknowledgments, and retransmission of lost packets [27]. This makes it suitable for applications where data integrity and order are paramount, though the overhead associated with these features can introduce latency, a well-documented trade-off for its reliability [20, 32]. While TCP provides strong reliability, its connection-oriented nature and overhead might introduce latency unsuitable for very time-critical, low-latency asynchronous communication between GALS modules in distributed IOPT systems.

2. **UDP** User Datagram Protocol provides a much simpler, connectionless service. It allows applications to send messages, known as datagrams, to other hosts without prior communication to set up special transmission channels [28]. UDP is considered an unreliable protocol as it does not guarantee delivery, order, or duplicate protection [20]. Its primary advantage is low latency due to the minimal protocol overhead, making it suitable for time-sensitive applications where occasional packet loss is acceptable [7]. The lightweight nature of UDP makes it highly attractive for efficient asynchronous communication between GALS modules in distributed IOPT sub-models, particularly where low latency is critical. Error handling and ordering, if required, would then need to be managed at a higher application layer within the IOPT sub-model's logic, leveraging the inherent flexibility of Petri nets to model such behaviors.

2.5 Design Flow for Distributed Systems: From Models to Networked Components

The development of distributed control systems, under the Globally Asynchronous, Locally Synchronous (GALS) paradigm [21, 4], often follows a structured design flow. This flow, illustrated in Figure 2.5, commences with a high-level system model and progresses through decomposition into components, eventually mapping these components onto specific implementation platforms with diverse network topologies. This process is fundamental to managing complexity and realizing distributed functionality, and it directly influences the selection of appropriate communication protocols.

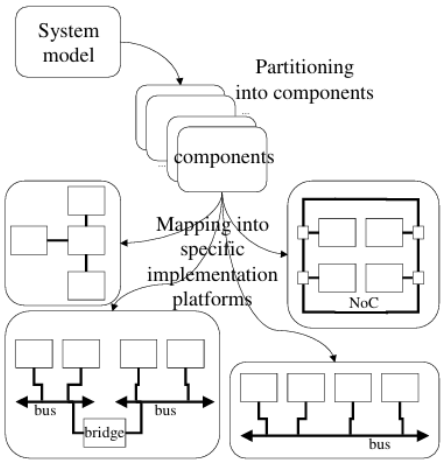

Figure 2.5: From models to code through model partitioning and mapping.

Conceptually, this design flow can be viewed in layers, as depicted in Figure 2.5:

- **Top Layer: System Model Definition**
  At the top is the initial system model, which encapsulates the overall behavior, concurrency, and functional requirements of the embedded system. Within the context of this work, Input-Output Place-Transition (IOPT) Petri nets serve as the primary focus for this system-level specification [14, 15]. This model represents the complete, monolithic view of the controller logic before any distribution is considered.

- **Middle Layer:Component Identification via Decomposition**
  The holistic system model, as shown in Figure 2.5, is then subjected to partitioning, where it is divided into a set of distinct, concurrent components or sub-models. Each component typically reflects a specific functionality or a logically separable part of the overall system. Within the IOPT-Tools framework, this partitioning is achieved through formal net decomposition operations, such as net splitting (detailed in Section 2.2.1) [13, 11]. This critical step transforms the single model into multiple IOPT sub-models, each representing a locally synchronous module designed to operate concurrently and interact with others. The "cutting sets" identified during decomposition define the logical interfaces and thus the necessary communication points between these emergent sub-models.

- **Bottom Layer: Component Mapping and Network Realization**
  Following decomposition, each sub-model (component) is mapped onto a specific implementation platform. These platforms can range from individual microcontrollers to dedicated processing units within a Field-Programmable Gate Array (FPGA), or even distinct software processes. As illustrated in Figure 2.5, the interaction between these distributed components must be realized through a defined

network structure. The choice of network topology is essential and can have different variations, including:

- **Direct Connections:** Point-to-point links between two components, often suitable for dedicated, high-throughput, or low-latency communication.
- **Shared Buses:** Several components share a common communication channel, such as a "bus" or a "bridge"-connected "multi-bus" system (Figure 2.5), which necessitates mechanisms (e.g., communication protocols) for arbitration and addressing.
- **Network-on-Chip (NoC) Structures:** As depicted in Figure 2.5, NoCs provide more complex, scalable, and often packet-based communication fabrics integrated onto a single chip, suitable for systems with many interacting components.

This layered approach directly correlates with the **GALS paradigm**. The decomposed IOPT sub-models from the middle layer naturally form the "Locally Synchronous" (LS) islands, each potentially operating within its own clock domain [21]. The communication between these locally synchronous systems (or LS islands), facilitated by the network topologies defined at the bottom layer, is inherently asynchronous, realizing the "Globally Asynchronous" (GA) nature of the system [4]. IOPT nets extended with Asynchronous Channels (ACs) and Time Domains (TDs) are specifically designed to model such GALS systems, where ACs represent the logical communication pathways between components operating in different TDs [21, 14].

The **IOPT net decomposition** (net splitting, Section 2.2.1) is thus a foundational step in this architectural mapping. It not only breaks down complexity but also explicitly defines the interfaces where inter-component communication is required. These interfaces, often realized as shared places or synchronized transitions in the original model, become the points where communication channels – logical at first, then physical – must be established [13].

Finally, the selection of **communication protocols** (as detailed in Section 2.4) is intrinsically linked to the chosen network topology and the characteristics of the GALS interactions. For instance:

- Direct connections might be implemented using UART for simple serial data exchange [38, 29] or a dedicated FIFO with handshake logic for high-speed, reliable data streaming between two specific FPGAs or modules (as discussed in Section 2.4.4).
- Shared bus topologies, as illustrated in Figure 2.5, naturally lend themselves to protocols like I^2C, which includes addressing for multiple devices [26, 36], or SPI, where multiple slaves can be managed via individual select lines [17, 19].
- NoC architectures often employ more sophisticated packet-based protocols that handle routing, flow control, and error checking internally.

The communication channels modeled in the IOPT GALS framework, representing events or data exchange between sub-models, must be implemented using these protocols over the selected physical network. The design of the proposed automated code generation tool, central to this dissertation, aims to bridge the gap between the high-level specification of these decomposed IOPT sub-models and the concrete implementation of the communication logic using these diverse protocols and their underlying topological assumptions.

2.6 IOPT tools

The theoretical constructs of IOPT Petri nets (Section 2.2) and the principles of GALS architectures (Section 2.3) find practical application in controller design through specialized software environments. One such comprehensive platform is IOPT-Tools, an integrated, web-based development environment tailored for the design, verification, and implementation of embedded system controllers, particularly for industrial automation and digital systems [14]. As previously mentioned [14], IOPT-Tools supports a model-driven development workflow, starting from graphical Petri net model creation to verification and, crucially for this work, automatic code generation in C or VHDL.

The environment's capacity to handle complex systems is partly due to its support for IOPT Petri net characteristics, including the input and output mechanisms essential for controller interaction and features facilitating model modularity, such as the net decomposition operations discussed in Section 2.2.1. These operations allow a complex controller model to be broken down into several distinct sub-models, which can then be targeted for execution on separate computational nodes, aligning with the distributed control paradigm.

In addition to design and verification, IOPT-Tools includes automatic code generation capabilities, enabling the creation of software in C or hardware descriptions in VHDL [12]. This feature streamlines the transition from design to deployment, allowing for efficient and error-free implementation of the controller in either software or hardware [18].

IOPT-Tools delivers a complete, start-to-finish solution for creating controllers for embedded systems. The integration of graphical design, formal verification, and automatic code generation within a single, web-based platform significantly increases efficiency, reliability, and speed of controller development for both industrial and digital applications.

2.6.1 Highlighting the Communication Gap

While IOPT-Tools provides this extensive support for developing and generating code for individual controller logic, and facilitates the decomposition of models for distributed architectures, a significant challenge remains in automatically establishing and managing the communication between these distributed sub-models. The current automatic code generation primarily focuses on the internal logic of each individual sub-model. Crucially, it does not extend to automatically generating the necessary communication infrastructure

required for these distinct sub-models to interact efficiently and reliably when deployed across different computational nodes.

This lack of automated support for different sub models, or petri nets, communication means that engineers must currently undertake the complex and error-prone task of manually implementing these communication links. This manual process involves selecting appropriate communication technologies (such as those reviewed in Section 2.4), writing and integrating low-level driver code, and ensuring data consistency. This not only diminishes the benefits of automated generation for the core logic but also increases development time, hinders system optimization, and introduces potential for integration issues.

Addressing this specific gap,by designing and implementing a tool that automates the generation of code for efficient and reliable communication channels between distributed IOPT sub-models, is the central objective of this dissertation. This will further enhance IOPT-Tools' capabilities for developing truly distributed control systems.

2.6.2 Overview of Key Components in IOPT-Tools

A central component of the IOPT-Tools is its **graphical editor**, which facilitates the interactive design and modification of IOPT Petri net models directly within a standard web browser [11]. This editor leverages AJAX principles, dynamically manipulating PNML (Petri Net Markup Language) data in an XML DOM document and utilizing XSL transformations to generate real-time SVG graphical representations. This architecture enables cross-platform compatibility and collaborative design, with features such as a persistent clipboard for inter-model data transfer and server-side sharing [11]. The editor rigorously enforces IOPT-net syntactic rules and includes a specialized expression editor that guides users in constructing valid mathematical expressions for guard functions and output actions, thereby minimizing syntax errors [11].

For system verification and debugging, IOPT-Tools incorporates a robust **verification engine** primarily composed of a state-space generator and a query system [18, 24]. The state-space generator computes the reachability graph of an IOPT model, identifying potential design flaws such as deadlocks and transition conflicts. While state-space graphs can be extensive, the tool provides statistics and the option to view the graph for smaller models [12]. To address the impracticality of visually inspecting large state-spaces, the query system allows users to define specific conditions based on net marking, output signals, and fired transitions. These queries are automatically checked during state-space computation, enabling automated model checking and property verification [24].

The framework also includes a **simulator tool** for executing and debugging IOPT models within the web browser [24]. Distinct from traditional Petri net simulators, the IOPT simulator is designed for non-autonomous systems, allowing users to manipulate input signals and autonomous input events directly. It supports step-by-step and continuous execution with programmable speeds and breakpoints. A key feature is the automatic

recording of simulation history, including net marking, signal values, and event triggers, which can be replayed, navigated, or exported for detailed analysis [24]. The simulator employs a compilation execution strategy, dynamically generating JavaScript code for model semantics to ensure high performance [24].

Regarding **code generation**, IOPT-Tools offers automatic tools to produce C and VHDL from IOPT models [18, 12]. The C code generator produces ANSI C files suitable for microcontrollers or PCs, with the `net_io.c` file requiring adaptation for specific hardware interfaces [18]. The VHDL code generator synthesizes VHDL component architectures, defining external interfaces based on IOPT signals and events, and handling internal logic for Petri net execution semantics [12]. These generated components are synchronous, requiring input signals to be synchronized with a clock [12]. While automatic code generation streamlines implementation and reduces low-level coding errors, the VHDL generation, in particular, may incur a small penalty in resource consumption compared to expert-coded designs, though hardware vendor optimization tools are effective in mitigating this [12].However, while these generators effectively produce code for the internal logic of individual IOPT sub-models, they currently do not automatically generate the necessary communication infrastructure or low-level driver code required for these distinct sub-models to interact efficiently and reliably when deployed across different computational nodes.

Although not fully integrated into the main web interface at the time of some publications, other supplementary tools exist. These include SnoopyIOPT (an alternative editor), Split (essential for model decomposition into sub-models, as discussed in Section 2.2.1), Animator (for animated synoptics and GUIs), GUI Generator for FPGA (for GUI code from Animator), Configurator (for I/O pin and hardware resource assignment), and HIPPO (for Petri net analysis and incidence matrix calculation) [18]. The potential for future integration of these tools into the web interface and the ongoing development of features like a waveform editor and in-circuit emulation highlight the continuous evolution of the IOPT-Tools framework [24, 18].

3

Work plan

The IOPT-Tools environment provides robust support for modeling, verifying, and generating code for individual controller sub-models using Input-Output Place-Transition (IOPT) Petri nets [14, 2, 15]. However, a notable challenge arises in distributed control systems, particularly those adhering to the Globally Asynchronous, Locally Synchronous (GALS) paradigm, where decomposed sub-models require inter-communication [21, 13]. As identified in Section 2.6.1, the existing automatic code generation primarily focuses on the internal logic of each sub-model, necessitating manual implementation of the communication links between them. This manual process is not only time-consuming but also highly prone to error, leading to significant challenges in debugging and system validation. This dissertation directly addresses this gap by proposing an automated code generation tool.

The core objective is to develop a mechanism that analyzes decomposed IOPT sub-models, OBTAIN THROUH DECOMPOSE GALS IN IOPT-TOOLS and automatically generates the necessary communication infrastructure code, thereby streamlining the development of distributed control systems and automating the creation of efficient and reliable data exchange pathways.

3.1 Overall Architecture of the Automated Generation Tool

This section provides an expanded characterization of the proposed solution, focusing on the architecture of the automated code generation tool. This tool, developed for integration within the IOPT-Tools ecosystem, is implemented as an API that translates abstract model specifications into executable code. Its architectural framework is systematically delineated by three primary components: the required inputs, the core transformation logic, and the resultant code outputs.

3.1.1 Inputs

The tool requires a set of specific artifacts to initiate the code generation process. These inputs provide the necessary information about the system's structure and desired behavior:

- **Decomposed IOPT Sub-models:** The primary input consists of the set of IOPT Petri net sub-models that result from the formal net splitting operation within IOPT-Tools.
- **Communication Configuration:** A set of user-provided parameters that specify the communication infrastructure. This input defines the desired communication protocol (e.g., I^2C, SPI, UART) and its corresponding settings (e.g., device addresses, bus speed, pin assignments).

3.1.2 Processing Logic

The core of the tool is its processing engine, which translates the abstract model into an implementation plan. This involves several steps:

- **Model Parsing:** The tool first parses the PNML representation of the IOPT sub-models to build an internal representation of the distributed system's structure.
- **Channel Identification:** It analyzes the interface definitions (cutting sets) and Time Domain (TD) annotations to identify all required communication channels. For example, it flags an interaction between a transition in TD1 and a transition in TD2 as a necessary communication link.
- **Protocol Mapping:** Based on the characteristics of each identified channel and user configuration, the tool maps the abstract communication requirement to the specific semantics of a chosen protocol. An event trigger might be mapped to a single command byte, while a data transfer might be mapped to a multi-byte packet.

3.1.3 Outputs

Upon completion of its processing logic, the tool generates a collection of software artifacts required to compile and deploy the distributed system:

- **Source Code:** The primary output is the C or VHDL source code that implements the communication logic for each sub-model.
- **Configuration Headers:** Automatically generated header files containing definitions for the system, such as slave addresses, command identifiers, and buffer sizes.
- **Integration Artifacts:** Potentially, the tool could also generate supporting files like makefiles or project files to further simplify the integration and compilation process.

3.2 Mapping Model Constructs to Implementation Primitives

A critical function of the generation tool is to create a conceptual bridge between the abstract formalism of the IOPT Petri net model and the concrete primitives of a programming language. This section describes how high-level constructs from the GALS-extended IOPT models are systematically translated into tangible programming constructs.

- A **synchronized transition** in the model, which represents a lock-step interaction, typically maps to a master-slave command pattern. For instance, the master component sends a specific command byte, and the slave component's logic waits to receive that exact byte before proceeding.
- An **asynchronous channel (AC)**, designed for communication across clock domains, naturally maps to a hardware-agnostic FIFO buffer coupled with a handshake protocol. This is often implemented with `valid/ready` signals to ensure data integrity without a shared clock.
- Data associated with a **token** residing in a shared place, representing a shared resource or data item, maps directly to the payload of a communication packet. The transfer of the token from one sub-model to another is realized by transmitting this payload.

3.3 Protocol Selection and Rationale

Before presenting a specific code example, it is important to briefly discuss the criteria governing the selection of an appropriate communication protocol. The automated tool is designed to either make an intelligent default selection or to guide the user in choosing a protocol based on the application's requirements. The selection is informed by the factors analyzed in the state of the art (Section 2.4):

- **Synchronicity Model:** The primary consideration is whether the communication is between locally synchronous modules or across globally asynchronous domains. Protocols like SPI and I^2C are well-suited for communication within a single, shared clock domain, whereas UART or FIFO-based logic are ideal for inter-domain GALS communication.
- **Network Topology:** The physical arrangement of the controllers influences the choice. A point-to-point link might favor UART, while a multi-drop bus connecting one master to several slaves makes I^2C or SPI (with multiple chip selects) a more logical choice.
- **Performance Characteristics:** The decision also involves trade-offs between latency, throughput, and reliability. For time-sensitive applications, a low-latency protocol

like SPI might be preferred, whereas for non-critical status updates, the simplicity of UART could be sufficient.

3.4 Implementation Case Study: An I²C-Based Channel

To demonstrate the tool's practical output, this section presents a case study focused on a common scenario in distributed control: triggering a remote event from one microcontroller to another.

For this example, a multi-slave bus architecture was assumed, making the I²C protocol an appropriate choice due to its built-in addressing scheme and low hardware pin count. The following subsection will detail the generated C code for the slave device, illustrating how the asynchronous reception of an I²C command is safely integrated into the synchronous execution loop of the IOPT controller.

3.5 I²C Communication Implementation

3.5.1 Library Utilization

The communication channel is implemented using the standard `Wire` library, which provides a high-level interface for configuring and operating the I²C bus on embedded controllers. This library supports both master and slave roles, synchronous data transmission, and interrupt-driven message reception.

3.5.2 Configuration

The I²C bus requires two signal lines and a common ground connection:

- **SDA (Serial Data):** GPIO 21
- **SCL (Serial Clock):** GPIO 22

Although these pin assignments correspond to the default mapping of the `Wire` library for the target platform, they are not explicitly defined in the generated code. Pull-up resistors are used on both lines in accordance with the I²C protocol specification to ensure reliable signal levels. The master device initializes the bus using `Wire.begin()` without an address parameter, whereas the slave device specifies its own address using `Wire.begin(I2C_SLAVE_ADDRESS)` to enable message reception.

3.5.3 Generated Input Definitions

In addition to library-based configuration, certain I²C parameters are provided by the code generation tools and embedded into the program as preprocessor directives. For instance:

```
#define I2C_SLAVE_ADDRESS      0x09   // Address assigned to this device
#define CMD_TRIGGER_EVENT9     0xA2   // Command linked to Petri net event
```

These constants ensure that device addressing and event-triggering commands remain consistent with the model-level specification from which the code is derived.

3.5.4 Message Transmission (Master Side)

Whenever a Petri net output event must be communicated to a slave device, the master initiates a write operation on the bus. To support communication with multiple slave devices, the message transmission logic is generalized into the following function:

```
// Generalized I²C write operation
void triggerMasterSendSlaveEvent(byte slaveAddress, byte command) {
    Serial.println("Attempting to send command to slave...");

    // Step 1: Begin a transmission to the I²C slave device
    Wire.beginTransmission(slaveAddress);

    // Step 2: Send the command byte
    Wire.write(command);

    // Step 3: Stop the transmission and send the data
    byte error = Wire.endTransmission();

    // Check the status of the transmission
    if (error == 0) {
        Serial.println("Command sent successfully!");
    } else {
        Serial.print("Error sending command. Error code: ");
        Serial.println(error);
        Serial.println("Check connections and slave address.");
    }
}
```

This function introduces two parameters:

- `slaveAddress` (`byte`): The 7-bit I^2C address of the destination slave device. This argument allows the master to dynamically target different devices connected to the bus, in accordance with the constants generated by the tool (e.g., `I2C_SLAVE_ADDRESS`).
- `command` (`byte`): The encoded instruction to be transmitted, typically representing an event identifier derived from the Petri net model (e.g., `CMD_TRIGGER_EVENT9`).

The function reports the transmission status via the return code of `Wire.endTransmission()`. A result of `0` indicates successful delivery, while non-zero values reflect communication errors (e.g., device not acknowledging the address or line disconnection). This feedback, logged through the serial interface, facilitates runtime debugging of bus integrity and device configuration.

3.5.5 Message Reception (Slave Side)

The slave controller employs an interrupt service routine (ISR) to process incoming messages. Upon receiving data, the ISR sets an internal flag to synchronize message handling with the Petri net execution cycle:

```
// Flag for signaling Petri net input
static volatile int event_trigger_flag = 0;

void receiveI2CEvent(int byteCount) {
    if (Wire.available() > 0) {
        char command = Wire.read();
        if (command == CMD_TRIGGER_EVENT) {
            event_trigger_flag = 1;  // Mark event for Petri net
        }
    }
    // ... clear buffer if required
}
```

In the main execution loop, this flag is polled and mapped to the Petri net input structure:

```
if (events != NULL) {
    if (event_trigger_flag == 1) {
        events->event = 1;         // Inject event into Petri net
        event_trigger_flag = 0;    // Reset flag after processing
    } else {
        events->event = 0;
    }
}
```

This design decouples the asynchronous arrival of I²C messages from the synchronous execution of the Petri net, ensuring deterministic and reliable event handling.

3.6 UART Communication Implementation

3.6.1 Library Utilization

The UART communication is implemented using the `HardwareSerial` class provided by the ESP32 framework. This library enables full-duplex serial communication over dedicated RX/TX pins, supporting multiple UART interfaces on the controller. The generated code employs a secondary hardware serial interface (`MySerial`) configured with a baud rate of 115200 and standard serial framing (`SERIAL_8N1`).

3.6.2 Input Definitions

Both the pin assignments and the UART message identifiers are defined by the user as inputs to the code generation tool. These inputs are automatically transformed into preprocessor definitions and constant values in the generated code, ensuring a direct mapping between the model-level specification and the executable program.
For pin assignments, the tool generates:

```
#define RXD2 16   // UART Receive pin
#define TXD2 17   // UART Transmit pin
```

For message identifiers linked to Petri net events, the tool generates:

```
const String MESSAGE_UART_EVENT43 = "TRIGGER_EVENT43";
const String MESSAGE_UART_EVENT52 = "TRIGGER_EVENT52";
```

This approach guarantees that the physical configuration (pins) and the logical communication events (messages) remain consistent across model specification and generated code.

3.6.3 Initialization

The initialization routine establishes both the debugging channel (`Serial`) and the application channel (`MySerial`):

```
void setupUart() {
    Serial.begin(115200);
    MySerial.begin(115200, SERIAL_8N1, RXD2, TXD2);
    Serial.println("UART communication initialized.");
}
```

3.6.4 Bidirectional Communication

Unlike the I^2C bus, UART does not distinguish between master and slave roles. Instead, both controllers can transmit and receive messages independently over their TX/RX lines.

The generated code therefore provides symmetric functions for sending and receiving messages.

Sending Messages. Transmission of events is achieved through the following function:

```
void sendDataUart(String message) {
    MySerial.println(message);
    Serial.print("Message sent: ");
    Serial.println(message);
}
```

This function allows Petri net output events to be directly mapped into UART message transmissions.

Receiving Messages. Reception is handled by continuously monitoring the UART buffer. When a valid message string is detected, it is matched against the generated identifiers and mapped to a Petri net event:

```
String receiveDataUart() {
    if (MySerial.available()) {
        String data = MySerial.readStringUntil('\n');
        data.trim();
        return data;
    }
    return "";
}

void waitMessageUart() {
    String receivedMessage = receiveDataUart();
    if (receivedMessage.length() > 0 &&
        receivedMessage == MESSAGE_UART_EVENT52) {
        event52_trigger_flag = true;
        Serial.println("Correct message received. Flag 52 activated.");
    }
}
```

The reception logic sets an internal flag (`event52_trigger_flag`), which can then be polled and mapped into the Petri net input structure. This design decouples asynchronous UART message arrivals from the synchronous execution cycle of the Petri net.

3.6.5 Integration with Petri Net Execution

Finally, the UART initialization is automatically embedded into the generated I/O configuration routine:

```
setupUart();  // Initialize UART communication
```

This ensures that both pin assignments and message identifiers, provided as inputs to the generation tool, are consistently integrated into the system before the execution of Petri net cycles. The resulting design enables fully bidirectional communication between controllers, where either side can trigger or respond to Petri net events.

3.7 TCP Communication with MQTT

3.7.1 Library Utilization

TCP/IP communication is implemented through the Wi-Fi and MQTT protocols using the `WiFi.h` and `PubSubClient.h` libraries. The Wi-Fi library provides network connectivity over IEEE 802.11, while the MQTT client library manages lightweight publish/subscribe communication over TCP. This design enables distributed controllers to exchange Petri net events through a broker-based infrastructure.

3.7.2 Input Definitions

All network and protocol parameters are defined by the user as inputs to the code generation tool. These include:

- **Wi-Fi credentials:** SSID and password for network access.
- **MQTT broker settings:** hostname/IP address and port number.
- **Client identifier:** a unique string for each device, e.g.,

  ```
  const char* client_id_TD_2 = "ESP32_IOPT_TD_2";
  ```

- **MQTT topics:** communication channels used for subscribing and publishing, e.g.,

  ```
  const char* topic_sub = "device/events/commands"; // Topic to subscribe
  const char* topic_pub = "device/events/commands"; // Topic to publish
  ```

- **Event messages:** strings associated with Petri net events, e.g.,

  ```
  const char* message_event40 = "Action_Triggered_By_Event_40";
  const char* message_event46 = "Action_Triggered_By_Event_46";
  ```

These parameters are automatically embedded into the generated code, ensuring consistency between the model specification and runtime communication.

3.7.3 Initialization

The initialization routine connects the device to the Wi-Fi network and configures the MQTT client:

```
void tcpMqttInitializeIO() {
    Serial.begin(115200);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("Wi-Fi connected!");

    client.setServer(mqtt_broker, mqtt_port);
    client.setCallback(callback);
}
```

The `callback` function is registered to process incoming MQTT messages and translate them into Petri net events.

3.7.4 Receiving Messages

Whenever a subscribed message arrives, the callback function is triggered. The payload is compared with the generated event message strings, and the corresponding flags are updated:

```
void callback(char* topic, byte* payload, unsigned int length) {
    String message;
    for (int i = 0; i < length; i++) {
        message += (char)payload[i];
    }
    if (String(topic) == topic_sub) {
        if (message.equals(message_event40)) {
            event40_trigger_flag = true;
        } else if (message.equals(message_event46)) {
            event46_trigger_flag = true;
        }
    }
}
```

The internal flags (`event40_trigger_flag`, `event46_trigger_flag`) decouple the asynchronous arrival of MQTT messages from the synchronous execution cycle of the Petri net.

3.7.5 Maintaining the Connection

The MQTT client requires periodic polling to ensure connectivity and process incoming messages:

```
void loopDelayTcp(const char* topic) {
    if (!client.connected()) {
        reconnect(topic);
    }
    client.loop(); // Keeps MQTT connection alive and processes new messages
}
```

The `reconnect()` function handles reconnections in case of broker or Wi-Fi failures, ensuring robust communication.

3.7.6 Publishing Messages

Petri net output events are mapped to MQTT publish operations by transmitting the corresponding predefined message strings to the broker topic defined by the user as input:

```
client.publish(topic_pub, message_event40);
```

This approach allows the generated code to remain fully generic: the publish topic can be configured per device through the code generation tool. One controller can broadcast event activations, while other controllers subscribed to the corresponding topic receive them and map them into their own Petri net execution cycle.

3.7.7 Communication Model

In contrast to I^2C (master/slave) and UART (peer-to-peer), MQTT follows a broker-based publish/subscribe model. Each controller can both publish and subscribe to topics, enabling fully distributed event communication. The use of predefined message identifiers ensures deterministic mapping between Petri net events and TCP/MQTT messages in the generated code.

4

Conclusion

This dissertation plan addressed a critical gap in the development of distributed control systems using IOPT Petri nets within the IOPT-Tools environment. While IOPT-Tools provides robust support for modeling, verifying, and generating code for individual controller sub-models, it lacked automated mechanisms for establishing communication between distributed sub-models, an essential requirement for systems designed under the GALS paradigm.

The research began by reviewing the theoretical foundations of Petri nets, GALS architectures, and relevant communication protocols, establishing a comprehensive context for the problem. Through this analysis, the limitations of current tools, particularly the need for manual implementation of communication between the controllers,were clearly identified. This motivated the primary objective: to design and develop an automated code generation tool capable of producing the necessary communication infrastructure for distributed IOPT sub-models.

The proposed solution was characterized by its ability to configure and generate the communication channel between two controllers. This method simplifies the development process, minimizes manual coding mistakes, and improves the scalability and maintainability of distributed control systems.

Key outcomes and contributions include:

- Design and Implementation of an automated code generation mechanism, integrated within the IOPT-Tools workflow, bridging the gap between high-level modeling and practical deployment.

- Validation and Testing through representative case studies, demonstrating the effectiveness and reliability of the generated communication infrastructure.

By automating the generation of communication channels code, this work significantly extends the capabilities of IOPT-Tools, empowering researchers and engineers to develop more complex, distributed control systems with greater efficiency and confidence. The tool developed here lays a strong foundation for future research and enhancements, such

as supporting additional communication protocols, optimizing generated code for specific platforms, and further integrating verification and simulation features.

Continued development in this direction will help realize the full potential of model-driven engineering in distributed embedded systems, reducing development effort while ensuring system correctness and robustness.

Bibliography

[1] ARM. *AMBA 4 AXI4-Stream Protocol Specification*. Protocol Specification IHI 0051A. ARM Limited, 2010-03 (cit. on p. 13).

[2] J. P. Barros et al. "From Petri Nets to Executable Systems: An Environment for Code Generation and Analysis". In: *International Conference on Informatics in Control, Automation and Robotics*. 2004. URL: https://api.semanticscholar.org/CorpusID:36288484 (cit. on pp. 7, 20).

[3] L. Bening and H. D. Foster. *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*. Boston, MA: Springer Science+Business Media, 2002 (cit. on p. 13).

[4] D. Bormann and P. Cheung. "Asynchronous wrapper for heterogeneous systems". In: *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. 1997, pp. 307–314. DOI: 10.1109/ICCD.1997.628884 (cit. on pp. 9, 14, 16).

[5] A. Costa and L. Gomes. "Petri net partitioning using net splitting operation". In: *2009 7th IEEE International Conference on Industrial Informatics*. 2009, pp. 204–209. DOI: 10.1109/INDIN.2009.5195804 (cit. on p. 8).

[6] A. Costa and L. Gomes. "Petri net Splitting Operation within Embedded Systems Co-design". In: *2007 5th IEEE International Conference on Industrial Informatics*. Vol. 1. 2007, pp. 503–508. DOI: 10.1109/INDIN.2007.4384808 (cit. on p. 8).

[7] B. A. Forouzan. *TCP/IP Protocol Suite*. 4th. New York, NY: McGraw-Hill, 2010 (cit. on p. 14).

[8] D. Fried et al. "Sequential Relational Decomposition". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 432–441. ISBN: 9781450355834. DOI: 10.1145/3209108.3209203. URL: https://doi.org/10.1145/3209108.3209203 (cit. on pp. 7, 8).

[9] C. Girault and W. Reisig, eds. *Application and Theory of Petri Nets: Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets, Strasbourg, 23–26 September 1980, Bad Honnef, 28–30 September 1981*. Vol. 52. Informatik-Fachberichte. Berlin, Heidelberg: Springer, 1982 (cit. on p. 4).

[10] L. Gomes. "On conflict resolution in Petri nets models through model structuring and composition". In: *INDIN '05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005.* 2005, pp. 489–494. DOI: 10.1109/INDIN.2005.1560425 (cit. on p. 7).

[11] L. Gomes. *Petri Nets Modeling and Distributed Embedded Controller Design*. Presentation at Mini Symposium 2014, Óbuda University. Available at https://conf.uni-obuda.hu/minisymp2014/LuisGomes_MiniSymposium_2014September2.pdf. 2014-09. URL: https://conf.uni-obuda.hu/minisymp2014/LuisGomes_MiniSymposium_2014September2.pdf (cit. on pp. 15, 18).

[12] L. Gomes et al. "From Petri net models to VHDL implementation of digital controllers". In: *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*. 2007, pp. 94–99. DOI: 10.1109/IECON.2007.4460403 (cit. on pp. 17–19).

[13] L. Gomes et al. "Merging and Splitting Petri Net Models within Distributed Embedded Controller Design". In: 2012-01, pp. 153–166. ISBN: 9781466639232. DOI: 10.4018/978-1-4666-3922-5.ch009 (cit. on pp. 8, 15, 16, 20).

[14] L. Gomes et al. "The Input-Output Place-Transition Petri Net Class and Associated Tools". In: *2007 5th IEEE International Conference on Industrial Informatics*. Vol. 1. 2007, pp. 509–514. DOI: 10.1109/INDIN.2007.4384809 (cit. on pp. 6, 7, 9, 15–17, 20).

[15] L. Gomes and J. P. Barros. "Refining IOPT Petri Nets Class for Embedded System Controller Modeling". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. 2018, pp. 4720–4725. DOI: 10.1109/IECON.2018.8592921 (cit. on pp. 15, 20).

[16] Hannes Siebeneicher. *Universal Asynchronous Receiver-Transmitter (UART)*. https://docs.arduino.cc/learn/communication/uart/. Accessed: 2025-04-28 (cit. on p. 13).

[17] hilscher. *SPI Interface*. https://www.hilscher.com/service-support/glossary/spi-interface. Accessed: 2025-05-25 (cit. on pp. 11, 12, 16).

[18] "IOPT Tools User Manual". In: URL: http://gres.uninova.pt. (visited on 2025-03-14) (cit. on pp. 17–19).

[19] keysight. *SPI Basics*. https://www.keysight.com/used/ph/en/knowledge/glossary/oscilloscopes/what-is-spi-the-basics-of-serial-peripheral-interface. Accessed: 2025-05-25 (cit. on pp. 12, 16).

[20] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. 8th. Hoboken, NJ: Pearson, 2021 (cit. on p. 14).

[21] F. Moutinho and L. Gomes. "Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems". In: vol. 372. 2012-01, pp. 143–150. ISBN: 978-3-642-28254-6. DOI: 10.1007/978-3-642-28255-3_16 (cit. on pp. 9, 10, 14, 16, 20).

[22] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143 (cit. on pp. 1, 4, 5).

[23] R. Pais, S. Barros, and L. Gomes. "A tool for tailored code generation from Petri net models". In: *2005 IEEE Conference on Emerging Technologies and Factory Automation*. Vol. 1. 2005, 8 pp.–864. DOI: 10.1109/ETFA.2005.1612615 (cit. on p. 7).

[24] F. Pereira and L. Gomes. "Cloud Based IOPT Petri Net Simulator to Test and Debug Embedded System Controllers". In: *Technological Innovation for Cloud-Based Engineering Systems*. Ed. by L. M. Camarinha-Matos et al. Cham: Springer International Publishing, 2015, pp. 165–175. ISBN: 978-3-319-16766-4 (cit. on pp. 6, 18, 19).

[25] C. A. Petri. "Kommunikation mit Automaten". English translation: "Communication with Automata", Technical Report RADC-TR-65-377, Griffiss Air Force Base, NY, 1966. PhD thesis. Bonn, Germany: Institut für Instrumentelle Mathematik, Universität Bonn, 1962 (cit. on pp. 1, 4).

[26] K. Petropoulos, G.-C. Vosniakos, and E. Stathatos. "ON FLEXIBLE MANUFACTURING SYSTEM CONTROLLER DESIGN AND PROTOTYPING USING PETRI NETS AND MULTIPLE MICRO-CONTROLLERS". English. In: vol. 19. 1. Copyright - Copyright University "Politehnica" of Bucharest, Machine and Manufacturing Systems Department and Association ICMAS 2024; Última atualização em - 2025-03-03. 2024, pp. 17–24. URL: https://www.proquest.com/scholarly-journals/on-flexible-manufacturing-system-controller/docview/3119238675/se-2 (cit. on pp. 11, 16).

[27] J. Postel. *Transmission Control Protocol*. RFC 793. Internet Engineering Task Force, 1981-09. URL: https://www.ietf.org/rfc/rfc793.txt (cit. on p. 14).

[28] J. Postel. *User Datagram Protocol*. RFC RFC 768. Internet Engineering Task Force, 1980-08. URL: https://www.rfc-editor.org/rfc/rfc768.html (cit. on p. 14).

[29] S. Rao and A. Fuller. *Low Voltage Translation For SPI, UART, RGMII, JTAG Interfaces*. Tech. rep. 2021. URL: www.ti.com (cit. on pp. 12, 13, 16).

[30] W. Reisig. *Understanding Petri nets. Modeling techniques, analysis methods, case studies. Translated from the German by the author*. 2013-07. ISBN: 978-3-642-33277-7. DOI: 10.1007/978-3-642-33278-4 (cit. on p. 5).

[31] M. Silva. *50 years after the PhD thesis of Carl Adam Petri: A perspective*. Tech. rep. URL: http://www.informatik.uni-hamburg.de/TGI/PetriNets/meetings/. (cit. on p. 5).

[32] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley Professional, 1994 (cit. on p. 14).

[33] R. Tian, N. Okazaki, and K. Inui. "The Mechanism of Additive Composition". In: *CoRR* abs/1511.08407 (2015). arXiv: 1511.08407. URL: http://arxiv.org/abs/1511.08407 (cit. on p. 7).

[34] M. Tiwari et al. "The Comprehensive Review: Internet Protocol (IP) Address a Primer for Digital Connectivity". In: *Asian Journal of Research in Computer Science* 17 (2024-07), pp. 178–189. DOI: 10.9734/ajrcos/2024/v17i7488 (cit. on p. 13).

[35] J. F. Wakerly. *Digital Design: Principles and Practices*. 4th. Upper Saddle River, NJ: Pearson Prentice Hall, 2006 (cit. on p. 13).

[36] Z. Wang, Y. Weng, and Q. Wu. "Related methods introduction and application analysis of I2C bus". In: *Applied and Computational Engineering* 13 (2023-10), pp. 275–281. DOI: 10.54254/2755-2721/13/20230747 (cit. on pp. 11, 16).

[37] Wikipedia. *Petri net*. https://en.wikipedia.org/wiki/Petri_net. Accessed: 2025-04-25 (cit. on p. 4).

[38] Wikipedia. *Universal asynchronous receiver-transmitter*. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter. Accessed: 2025-04-28 (cit. on pp. 12, 16).