DUARTE JOSÉ RIBEIRO TAVARES

BSc in Electrical and Computer Engineering

COMMUNICATION MODULES FOR DISTRIBUTED CONTROLLERS SPECIFIED THROUGH IOPT MODELS

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

CONTENTS

List of Figures

Listings

iv

Acronyms

GALS Globally Asynchronous, Locally Synchronous *(p. 20)*

IOPT Input-Output Place-Transition *(pp. 20, 22, 23, 37, 40)*

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.

Transitions   Places   Arcs   Place with token
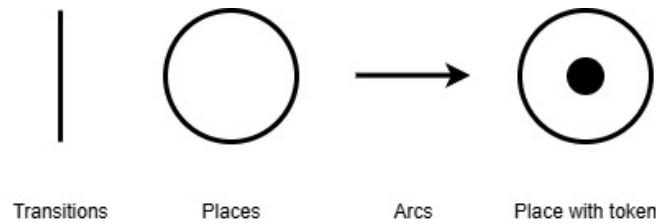
Figure 2.1: Petri net elements.

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.
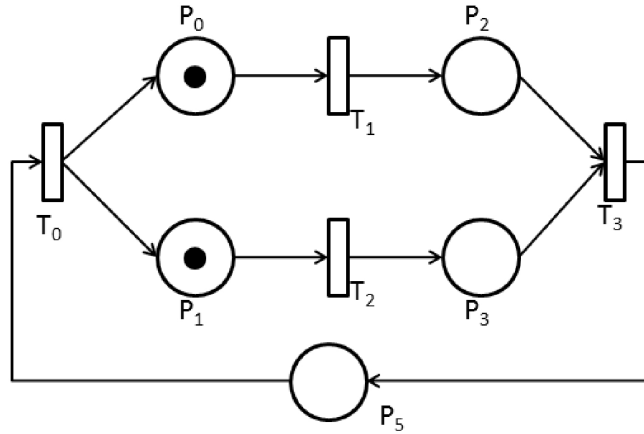
Figure 2.2: Example of Place/Transition net.

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].
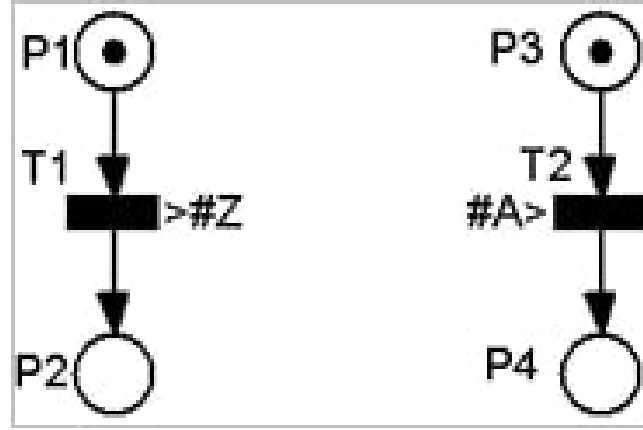
Figure 2.3: Example of IOPT Petri net.

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].

8

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:

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

9

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

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

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.
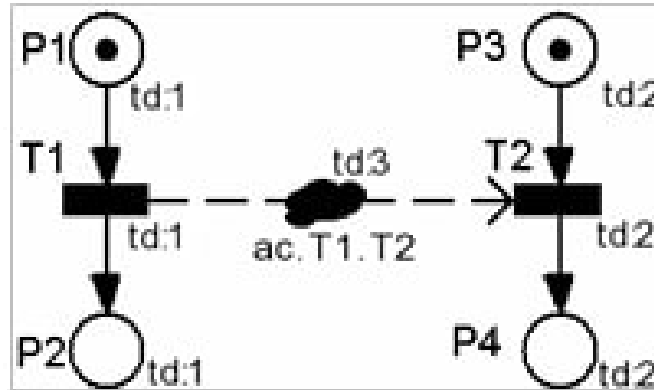
Figure 2.4: Example of Petri net with GALS system.

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 I²C

The I²C (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 I²C protocol presents certain limitations that may hinder its suitability for high-performance or large-scale applications. The standard I²C 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 I²C 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.
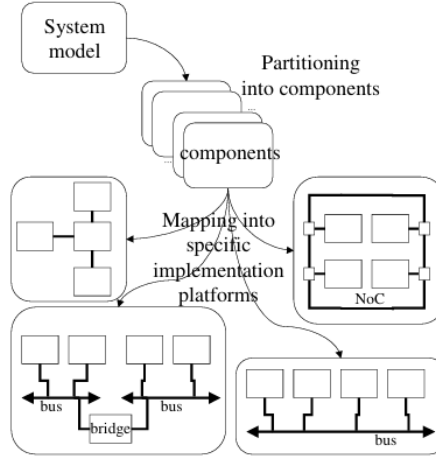
Figure 2.5: From models to code through model partitioning and mapping.

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

15

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].

System Design and Implementation

As discussed in Section 2.6.1, the IOPT-Tools environment provides robust support for modeling, verifying, and generating code for individual controller sub-models specified with Input-Output Place-Transition (IOPT) Petri nets [14, 2, 15]. However, a significant limitation arises in distributed control systems, particularly under the Globally Asynchronous, Locally Synchronous (GALS) paradigm, where decomposed sub-models require inter-communication [21, 13]. Current automatic code generation within IOPT-Tools focuses primarily on the internal logic of each sub-model, leaving the implementation of communication links to be carried out manually. This process is time-consuming, error-prone and poses substantial challenges for debugging and validation.

The objective of this dissertation is to address this gap by proposing an automated code generation tool capable of analyzing decomposed IOPT sub-models obtained through the decomposition of GALS systems in IOPT-Tools and automatically producing the necessary communication infrastructure code. In doing so, the tool streamlines the development of distributed control systems and ensures efficient and reliable data exchange pathways.

This chapter presents the design, implementation, and validation of the proposed tool. It begins with an overview of the system architecture, developed as a web-based Application Programming Interface (API) integrated within the IOPT-Tools ecosystem. The chapter then details the transformation pipeline, from model inputs to generated code outputs, and provides a formal specification of the API endpoints and parameters. Subsequently, the generated C++ code is analyzed with respect to three communication protocols: I²C, UART, and TCP/MQTT. Finally, the chapter outlines the validation methodology used to assess both the correctness and reliability of the tool's output.

3.1   Key Design Decisions

The tool is implemented as a server-side web Application Programming Interface (API). This architecture decouples the code generation logic from the end-user's local environment, providing a platform-independent and highly accessible solution. The core of the system is a single PHP script, `gals.php`(mudar), which resides on a web server and is

responsible for processing all incoming requests.

1. **User Request:** The user constructs a Hypertext Transfer Protocol (HTTP) GET request that contains a set of URL parameters. These parameters specify the desired communication protocol and its configuration, as detailed in Section 3.4.

2. **Server-Side Processing:** The web server receives the request and forwards it to the `gals.php`(mudar) script. The script parses and validates the input parameters, selects the appropriate code template based on the chosen protocol, and dynamically populates it with the user-provided values.

3. **Code Generation:** The script generates the complete C++ source code tailored to the user's specifications. In the event of invalid or missing parameters, the script generates an appropriate HTTP 400 error response.

4. **HTTP Response:** The server sends the generated code back to the user within an HTTP response, typically rendered as plain text in the browser.

Figure 3.1: The architectural workflow of the code generation API, from user request to the generated code response.(mudar)

3.2 Design Rationale

A Web API was chosen over a standalone desktop application or a command-line tool primarily for its inherent **platform independence** and **accessibility**. This approach eliminates the need for users to install any specialized software; the tool can be accessed from any device with a web browser or a standard HTTP client. This aligns with the web-based nature of the IOPT-Tools environment itself.

The choice of PHP as the implementation language for the proposed tool was motivated by several considerations. First, PHP is inherently designed as a server-side scripting language, well suited for handling web requests and generating dynamic content, which constitutes the core functionality of the tool. Second, since code generation is fundamentally a text manipulation task, the extensive library of built-in string processing functions of PHP greatly facilitated the development of the code templating engine. Third, PHP remains one of the most widely deployed back-end languages, with near-universal support across web hosting providers, thereby ensuring that the tool is both easily deployable and maintainable. Finally, the decision aligns strategically with the IOPT-Tools ecosystem, which itself is structured as a comprehensive web-based development environment. The adoption of a prominent server-side technology such as PHP thus maximizes architectural compatibility and establishes a clear path toward seamless integration of the code generation tool into the main IOPT-Tools web interface.

Although modern APIs often employ POST requests with JSON payloads, a simpler GET-based interface was selected for this application, as it offers several advantages particularly suited to the intended use case. First, GET requests can be constructed and tested directly in a web browser's address bar, greatly facilitating debugging, demonstration, and user adoption; this approach is explicitly supported through the tool's dedicated help page (`help.php`). Second, since each code generation request represents an independent and self-contained transaction, the stateless nature of GET requests provides a natural fit, eliminating the need to maintain server-side session state between consecutive requests. Finally, a complete API call, including all its parameters, is encapsulated within a single URL. This feature enhances usability by allowing users to easily save, bookmark, and share specific code generation configurations.

3.3 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 decomposed IOPT Petri net model and concrete primitives of the generated C++ code. This section describes how the API systematically translated the high-level constructs of the GALS-extended IOPT models into tangible programming constructs.

An important mapping concerns event-triggered transitions, which correspond to protocol-specific message transmissions. In the decomposed model, a transition that sends an event to another sub-model is implemented as a conditional block within the sender's `PutOutputSignals()` or `OutputSignals()` function. The activation of this transition, represented in code as `if (events->eventName == 1)`, is directly associated with a function call that transmits a message using the chosen protocol primitive. Examples include `Wire.write()` for I²C, `MySerial.println()` for UART, and `client.publish()` for MQTT.

Another essential mapping involves the Asynchronous Channel (AC), which in the abstract model facilitates communication across different clock domains in a GALS system. In software, this construct is realized through a two-part mechanism. First, an asynchronous reception primitive is used and its form depends on the chosen protocol: an Interrupt Service Routine (`onReceive`) for I²C, polling of the serial buffer (`MySerial.available()`) for UART, or a client callback function for MQTT. Second, upon receiving the correct message, this mechanism sets a `volatile` boolean flag. The flag functions as a safe, memory-mapped bridge between the asynchronous communication domain (where the message arrives) and the synchronous Petri net execution domain, where it is polled within the `GetInputSignals()` function.

Finally, the transfer of tokens between sub-models is directly mapped to the exchange of message payloads. Concretely, the data or state represented by a token is encoded in the `slaveMessage`, `uartMessage`, or `tcpMessage` parameter of the API. The successful

reception of this specific payload by the destination controller effectively completes the token transfer and enables the progression of the corresponding Petri net.

Through these mappings, the tool establishes a systematic and consistent correspondence between abstract model constructs and concrete implementation primitives, thereby ensuring that the semantics of GALS extended IOPT models are faithfully preserved in the generated code.

3.4 Application Programming Interface (API) Specification

This section provides a formal specification of the Application Programming Interface (API) for the code generation tool. The API is designed to be accessed via HTTP GET requests, with all configuration options passed as URL parameters.

The base endpoint for the API corresponds to the server location of the main script, as illustrated below:

```
http://<server_address>/gals.php
```

Every API request must include three global parameters that define the context for the code generation. These are detailed in Table 3.1.

Table 3.1: Global API Parameters

Parameter	Status	Description
protocol	Mandatory	Specifies the communication protocol. Must be one of: i2c, uart, or tcp.
projectName	Mandatory	A C-identifier compliant string defining the project name, used for function naming in the generated code.
eventName	Mandatory	A C-identifier compliant string defining the specific event to be handled.

3.4.1 Protocol-Specific Parameters

In addition to the global parameters, each protocol requires or accepts a unique set of parameters to configure its behavior according to the protocol needs.

I²C Protocol. For I²C communication, the parameters specified in Table 3.2 are required.

Table 3.2: API Parameters for the I²C Protocol

Parameter	Status	Type	Description
slaveAddress	Mandatory	Integer	The 7-bit address of the slave device. Valid range: 8–119.
slaveMessage	Mandatory	Char / Int	The command byte, which may be specified either as a single ASCII character or as an integer value (0–127).

UART Protocol. The UART protocol interface offers several optional parameters for fine-tuning the serial communication, as detailed in Table 3.3.

Table 3.3: API Parameters for the UART Protocol

Parameter	Status	Type	Description / Default Value
serialPort	Optional	Integer	The hardware serial port to use on the ESP32 (0, 1, or 2). **Default: 2**.
rxPin	Optional	Integer	The GPIO pin number for receiving data (RX). **Default: 16**.
txPin	Optional	Integer	The GPIO pin number for transmitting data (TX). **Default: 17**.
baudRate	Optional	Integer	The data transmission rate in bits per second. **Default: 115200**.
uartMessage	Optional	String	The message string that triggers the event. **Default: "trigger_<eventName>"**.

TCP/MQTT Protocol. For network communication via TCP/MQTT, the API requires a topic to be specified. Other parameters related to network and broker configuration are optional, with predefined default values that facilitate rapid prototyping and testing. These are listed in Table 3.4.

Table 3.4: API Parameters for the TCP/MQTT Protocol

Parameter	Status	Type	Description / Default Value
topic	Mandatory	String	The MQTT topic for publishing and subscribing.
clientID	Optional	String	The unique client identifier for the MQTT connection. **Default: "ESP32_IOPT"**.
broker	Optional	String	The address of the MQTT broker. **Default: "broker.hivemq.com"**.
port	Optional	Integer	The network port for the MQTT broker. **Default: 1883**.
ssid	Optional	String	The Service Set Identifier (SSID) of the Wi-Fi network. **Default: "yourNetworkName"**.
password	Optional	String	The password for the Wi-Fi network. **Default: "yourNetworkPassword"**.
tcpMessage	Optional	String	The message payload that triggers the event. **Default: "trigger_<eventName>"**.

3.5 Analysis of Generated Code

This section presents and analyzes the concrete C++ source code produced by the implemented API. For each of the three supported protocols, a sample API call is provided, followed by an analysis of the generated code's structure and key features. The full, verbatim source code for each example is available in the appendices for complete reference.

3.5.1 I²C-Based Communication Channel

The I²C protocol is well-suited for communication on a shared bus, following a master-slave paradigm. The API generates code for both master and slave controllers to facilitate this model.

To generate the code for a project named 'DemoProject' handling an event named 'buttonPress', the following API call is made:

```
?protocol=i2c&projectName=DemoProject&eventName=buttonPress&slaveAddress=9&slaveMessage=P
```

The complete C++ source code generated by the API call is provided in **Appendix A**.

A key feature of the slave controller's implementation is the safe handling of asynchronous events. To communicate an event from the hardware-triggered Interrupt Service Routine (ISR) to the main synchronous execution loop, a `volatile` boolean flag is employed. This critical design pattern, which prevents compiler optimizations that could lead to race conditions, is illustrated in the following snippet:

```
// Snippet from Slave Controller Code (Full code in Appendix \ref{appendix:i2c})

// PART 1: Defined in the global scope
volatile bool buttonPress_trigger_flag = 0;

// ...

// PART 3: Polled within the synchronous DemoProject_GetInputSignals()
if (buttonPress_trigger_flag == 1) {
    events->buttonPress = 1;
    buttonPress_trigger_flag = 0; // Reset the flag after processing
}
```

This pattern ensures that the event is processed reliably and decouples the asynchronous hardware interrupt from the synchronous Petri net execution cycle. The master controller code is more straightforward, with a dedicated function that encapsulates the I²C write operation (see Appendix A for the complete listing).

3.5.2 UART-Based Communication Channel

UART communication is ideal for point-to-point serial data exchange and does not rely on a shared clock. The generated code supports bidirectional communication between two peer controllers.

Sample API Call.

```
?protocol=uart&projectName=SystemA&eventName=toggleLED&baudRate=9600
```

The complete source code generated for this UART example can be found in **Appendix B**.

The implementation creates a `HardwareSerial` object for communication, preserving the primary serial port for debugging [cite: 594-596]. The code is symmetric, providing functions for sending and receiving data. Reception is handled by polling the serial buffer. When a valid message is received, its content is compared against a predefined constant. A match sets a `volatile` flag to signal the event to the synchronous `SystemA_GetInputSignals()` function, effectively decoupling the asynchronous serial input from the model execution.

3.5.3 TCP/MQTT-Based Communication Channel

For networked systems, the tool generates code utilizing the MQTT protocol over TCP/IP. This enables a robust publish-subscribe communication model managed by a central broker.

Sample API Call.

```
?protocol=tcp&projectName=SensorNetwork&eventName=alert&topic=sensors/events
```

The complete listing of the generated code for this TCP/MQTT example is available in **Appendix C**.

The code leverages the `PubSubClient.h` library to manage the MQTT connection [cite: 653]. The logic is centered around a `callback` function, which is triggered by the library upon receiving a message on a subscribed topic [cite: 677-678, 682]. Inside the callback, the message payload is compared to an expected event string. A match sets a `volatile` flag for the synchronous `SensorNetwork_GetInputSignals()` function. The code also includes a `reconnect` function to automatically re-establish the connection to the broker if it is lost, enhancing the system's resilience.

3.6 Tool Validation

Following the implementation of the code generation tool, a two-stage validation process was conducted to verify its reliability and functional correctness of its output. The first

stage focused on testing the API's behavior, while the second stage involved empirically validating the generated C++ code on target hardware.

3.6.1 API Functionality Testing

The API endpoint was systematically tested to ensure its robustness and adherence to the specification described in Section 3.3. The testing methodology involved subjecting the API to a series of HTTP GET requests with valid and invalid parameter sets.

The API was tested with a range of valid parameter combinations for the three supported protocols (I²C, UART, and TCP/MQTT). These tests confirmed that the tool consistently produced syntactically correct C++ code that matched the expected output for the given inputs.

To verify the error handling capabilities of the tool, a series of negative tests were performed. These included:

- Submitting requests with missing mandatory parameters (for example, omitting the `topic` for the TCP protocol).

- Providing values outside the specified valid range (for example, an I2C `slaveAddress` of 150).

- Using an unsupported protocol name in the `protocol` parameter.

In all such cases, the API correctly identified the input as invalid and returned the expected `HTTP 400` status code along with a descriptive error message, as designed. This confirmed the robustness of the input validation and error handling logic.

3.6.2 Generated Code Validation

The second stage of validation focused on confirming that the code generated by the tool was not only syntactically correct, but also functionally operational. For this purpose, a physical testbed consisting of two ESP32 micro-controller development boards was established for this purpose.

For each of the three protocols, the following procedure was executed:

1. A sample API call was made to generate the code for both the sending (Master/Publisher) and receiving (Slave/Subscriber) controllers.

2. The respective generated C++ code was integrated into a minimal Arduino project, compiled, and uploaded to the two ESP32 boards.

3. The boards were connected according to the requirements of the protocol being tested (for example, via I2C bus lines, cross-wired UART pins, or connected to the same Wi-Fi network for MQTT).

4. The sending device was triggered to activate the Petri net output event, and the receiving device's serial monitor was observed for confirmation of message reception.

   The empirical tests were successful for all three communication protocols. In each test case, the receiving device correctly registered the event triggered by the sending device. This validation process confirmed that the API generates functionally correct, reliable, and deployable communication modules that successfully implement the behavior specified in the IOPT model.

4

Case Study and Performance Analysis

4.1 Introduction

Having detailed the design and implementation of the automated code generation tool in Chapter 3, this chapter now seeks to validate its practical utility and analyze its output in a real-world scenario. Although the previous chapter verified the tool's functionality, this chapter demonstrates its application within the intended model-driven development workflow, from a high-level system specification to a deployed multi-controller system [cite: 1, 2, 3].

To achieve this, a case study is presented centered on a distributed controller for a three-conveyor automation system [cite: 3]. This application was chosen because it represents a common class of problems in industrial automation, involving multiple coordinated subsystems. The implementation of this case study serves two primary objectives, directly adapted from the research goals presented in [cite: 3]:

1. **To validate the end-to-end development workflow:** This involves demonstrating how the IOPT-Tools framework and the newly developed API are jointly employed to support low-code development of a distributed system. The process begins with a global IOPT model, proceeds through model decomposition, and culminates in the automatic generation and integration of both the controller logic and heterogeneous communication modules [cite: 3].

2. **To conduct a quantitative performance analysis:** By implementing intercontroller communication using three distinct protocols, I2C, UART, and TCP/MQTT. This chapter provides an empirical comparison of their respective resource overheads.]The analysis focuses specifically on the memory footprint, offering valuable insight into the trade-offs associated with each protocol in a resource-constrained embedded environment [cite: 3].

This chapter begins by describing the conveyor system use case and its corresponding IOPT model. Then it details the implementation process, showing how the automated tool was used to generate the necessary communication code. Subsequently, the results

of the performance analysis are presented and discussed. The chapter concludes with a summary of the key findings of the case study.

4.2 Use Case Description: The Three-Conveyor System

To demonstrate the practical application of the development workflow, a case study from the industrial automation domain was selected. The system under consideration is a controller for a three-conveyor belt set-up, designed to transport items sequentially from an entrance point to an exit point.

As illustrated in Figure 4.1, the physical system consists of three different conveyors. Each conveyor is equipped with two sensors to detect the presence of an item: one at its entrance (in1, in2, in3) and one at its exit (out1, out2, out3). An additional sensor (in4) detects when an item is removed from the end of the final conveyor. Control actions involve activating the motor for each of the three conveyors (move1, move2, move3). The objective of the controller is to ensure that the items move smoothly throughout the system, and each conveyor is activated only when an item is ready to be transferred and the subsequent conveyor is clear.
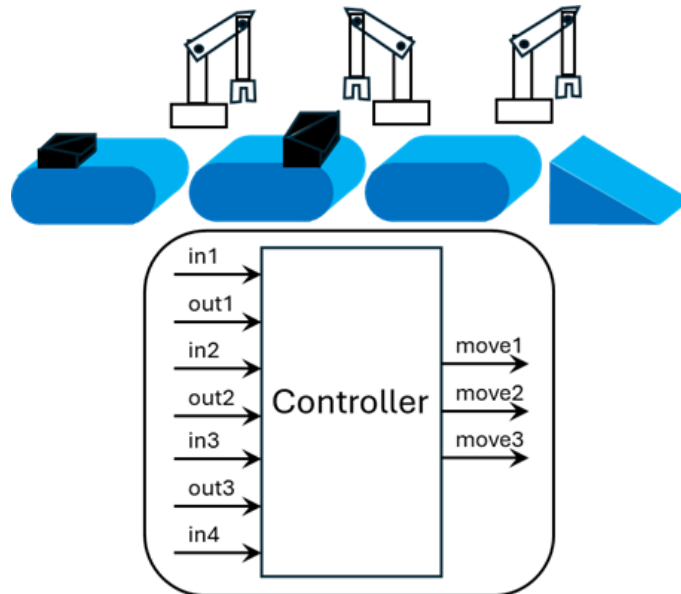
Figure 4.1: Layout of the three-conveyor system, showing the placement of sensors and the direction of item flow.

The complete centralized behavior of the controller is formally specified using the single global IOPT Petri net model shown in Figure 4.2. For clarity and brevity in this case study, the model assumes that each conveyor has a capacity of a single item. This global model represents the entire system logic before any consideration of a distributed implementation.

For deployment onto a set of distributed hardware controllers, the global model was decomposed using the net splitting operation available in IOPT-Tools. A cutting set,

Figure 4.2: The global IOPT Petri net model specifying the centralized controller for the entire conveyor system. The nodes highlighted in red represent the chosen cutting set for decomposition.

consisting of three places and three transitions (highlighted in red in Figure 4.2), was selected to partition the model along the physical boundaries of the three conveyors. The result of this operation is shown in Figure 4.3. The original centralized model is transformed into a set of four interconnected, concurrent sub-models, each responsible for a portion of the system (Entrance, Conveyor Two, Conveyor Three, and Exit). The

connections between these submodels represent the abstract communication channels that must be implemented to ensure the correct synchronized behavior of the distributed system.

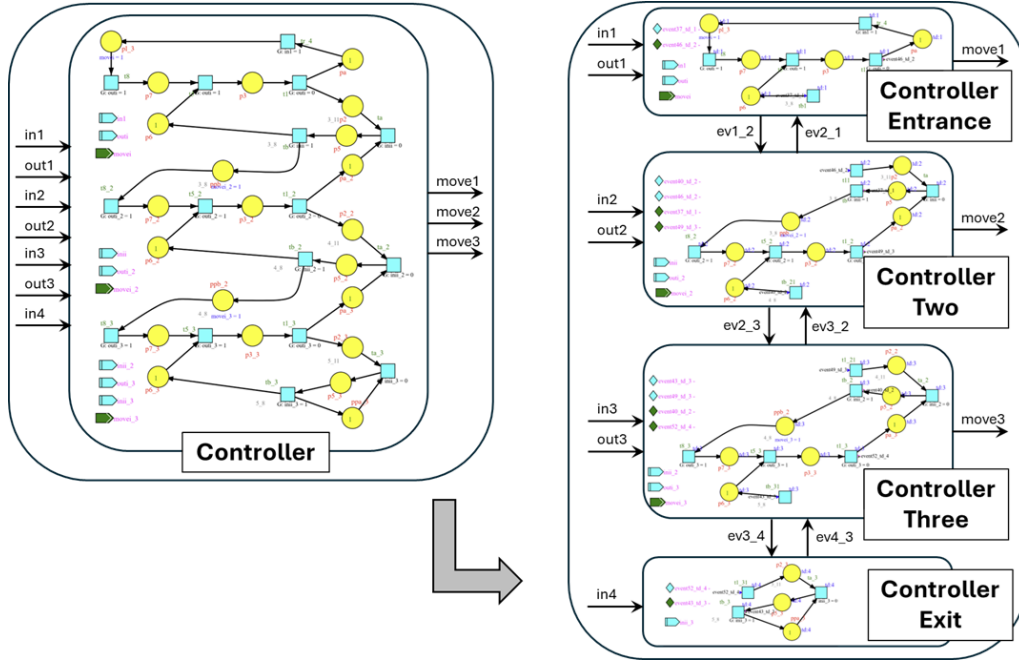Figure 4.3: The transformation from the centralized controller model (left) to a set of concurrent, networked sub-models (right) resulting from the net splitting operation.

4.3 Implementation via Automated Code Generation

With the global IOPT model decomposed into a set of concurrent sub-models (as shown in Figure 4.3), the next step in the development workflow is to implement the abstract communication channels between them. This task was accomplished using the automated code generation API detailed in Chapter 3.

The target deployment architecture for this case study, shown in Figure 4.4, employs a heterogeneous mix of three different communication protocols to connect the four controller modules. The following subsections detail how the specific code for each communication link was generated using the API.

4.3.1 I²C Bus Implementation

As depicted in the deployment architecture, *Controller Two* acts as the master on a local I²C bus, coordinating with *Controller Entrance* and *Controller Three*, which act as slaves. To generate the necessary master and slave code for an event representing a piece transfer, an API call similar to the following was used:

```
?protocol=i2c&projectName=ControllerTwo&eventName=TransferReady&slaveAddress=8&slaveMessa
```
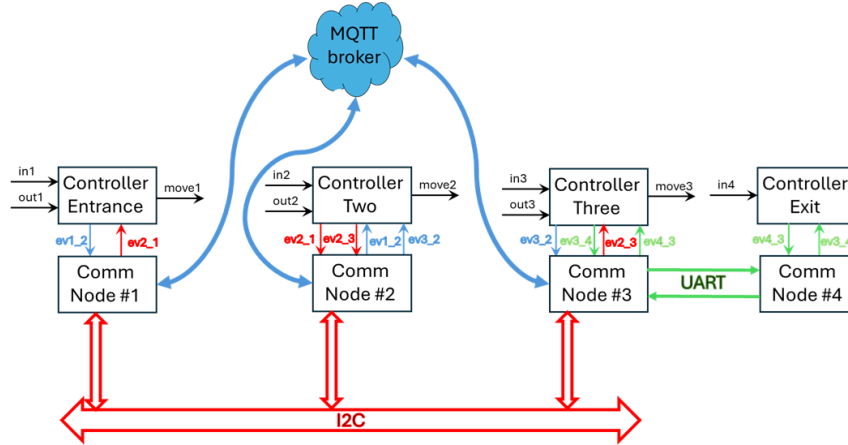
Figure 4.4: The heterogeneous deployment architecture for the distributed conveyor system controller, utilizing I²C, UART, and MQTT over TCP/IP for inter-controller communication and synchronization.

4.3.2 UART Point-to-Point Link

A direct, asynchronous serial link between *Controller Three* and *Controller Exit* was required to signal the final transfer of a piece. The code for this UART-based communication was generated with the following API call:

```
?protocol=uart&projectName=ControllerThree&eventName=PieceExiting&baudRate=115200
```

4.3.3 MQTT Network Communication

To handle higher-level, asynchronous status updates between the main controllers, the MQTT protocol was employed over a Wi-Fi network. For example, to allow *Controller Entrance* to publish a "Part Arrived" event to a central topic that *Controller Two* subscribes to, the following API call was made:

```
?protocol=tcp&projectName=ControllerEntrance&eventName=PartArrived&topic=conveyor/status
```

4.3.4 Code Integration

For each controller, the specific communication modules generated by the API calls were integrated with the logic code of the baseline controller produced by the standard IOPT-Tools generator. The entire process of specifying, generating, and integrating the code for all three heterogeneous communication links was completed in a fraction of the time required for manual implementation. This successfully demonstrates the tool's primary objective: to accelerate the development of distributed control systems and reduce the potential for implementation errors by automating the creation of communication infrastructure.

4.4 Performance Analysis and Results

After successfully implementing the distributed controller for the conveyor system, a performance analysis was performed to quantify the resource overhead associated with the different communication protocols generated by the tool. Although multiple performance metrics such as latency and CPU usage could be considered, this analysis focuses on the **memory footprint**, as it is often the most critical constraint in embedded systems development.

4.4.1 Methodology

The memory footprint of each of the four controller modules (Entrance, Two, Three and Exit) was measured to determine the overhead of its communication stack. The process was as follows.

1. **Baseline Measurement:** For each controller, the code containing only the Petri net execution logic (generated by the standard IOPT-Tools) was compiled for the ESP32 target. The static program storage space (Flash memory) reported by the compiler was recorded as the baseline.

2. **Measurement with Communication:** The API-generated communication code, corresponding to the heterogeneous architecture shown in Figure 4.4, was integrated with the baseline logic for each controller. The project was then recompiled and the new total program size was recorded.

3. **Overhead Calculation:** The communication overhead was calculated as the difference in memory footprint between the baseline measurement and the measurement with the integrated communication code.

4.4.2 Results

The results of the memory footprint analysis are summarized in Table 4.1. The data reveals a significant variation in overhead that is strongly correlated with the complexity of the communication protocols implemented on each controller module.

Table 4.1: 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

4.4.3 Discussion of Results

The data presented in Table 4.1 clearly illustrate the critical trade-off between functional flexibility and resource consumption when selecting a communication protocol.

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

This is further corroborated by the data from the **Exit Controller**. This module, which implements *only* the UART protocol for simple point-to-point asynchronous communication, shows a remarkably low overhead of just 1,420 bytes (a mere 0.15% increase). This stark contrast highlights the lightweight nature of hardware-level serial protocols compared to their network-level counterparts, as the former can be implemented with minimal software abstraction.

These findings have significant implications for the design of distributed embedded systems. The model-driven development approach, augmented by the tool presented in this thesis, allows a designer to make informed decisions during the system partitioning phase. 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 network connectivity. Simpler nodes can be implemented with lightweight protocols like UART or I2C, thus optimizing the overall system's resource usage and ensuring that hardware constraints are met.

4.5 Chapter Summary

This chapter successfully demonstrated the practical application and value of the automated code generation tool through a detailed case study of a three-conveyor distributed control system. The work presented herein validated the entire model-driven development workflow, beginning with a high-level IOPT Petri net model and culminating in a functional, multi-controller hardware implementation. The tool was shown to seamlessly integrate into this workflow, allowing the rapid and reliable implementation of a complex and heterogeneous communication architecture that involves the I2C, UART and TCP / MQTT protocols.

Furthermore, the performance analysis provided critical, quantitative insights into the resource consumption of these protocols. The results empirically confirmed the significant memory overhead of network-level protocols like TCP/MQTT when compared to the lightweight nature of hardware-level protocols such as UART. Ultimately, this case study validates the central proposition of this thesis: that automated generation of

communication modules not only accelerates development, but also empowers designers to make informed, data-driven decisions when balancing functional requirements against the hardware constraints of embedded systems.

5

CONCLUSION

This dissertation addressed a critical challenge in the model-driven development of distributed control systems: the lack of automated support for generating inter-controller communication infrastructure within the IOPT-Tools framework. Manual implementation of these communication links was identified as a time-consuming and error-prone process that hinders the rapid prototyping of distributed systems specified with IOPT Petri nets.

To bridge this gap, a server-side software tool was designed and implemented as a web-based Application Programming Interface (API). As detailed in Chapter 3, the tool is capable of dynamically generating C++ communication modules for three distinct protocols—I²C, UART, and TCP/MQTT—based on a simple set of URL parameters, thereby automating a previously manual task.

The practical utility and performance of this tool were subsequently validated through the comprehensive case study of a three-conveyor automation system, presented in Chapter 4. This study demonstrated the tool's successful integration into an end-to-end development workflow, from a high-level IOPT Petri net model to a functional, multi-controller hardware implementation. Furthermore, the case study provided a quantitative analysis of the memory footprint overhead associated with each protocol, offering empirical data on the trade-offs between them.

5.1 Main Contributions

The research and development detailed in this dissertation have resulted in several key contributions to the field of model-driven development for distributed embedded systems.

The first and most significant contribution is the design and implementation of a functional web-based API that automates the generation of communication code for distributed controllers specified with IOPT Petri nets. The tool supports three widely used protocols (I²C, UART, and TCP/MQTT) and directly addresses a critical gap in the IOPT-Tools framework, replacing a manual, error-prone task with a rapid and reliable automated process.

A second contribution is the validation of an end-to-end model-driven workflow.

This dissertation demonstrates how a high-level system model can be decomposed and subsequently implemented on a physical, multi-controller testbed by integrating the code generated by IOPT-Tools with the communication modules produced by the new API. The successful implementation of the heterogeneous conveyor system case study provides concrete evidence of the feasibility and effectiveness of this workflow.

A final contribution is the novel empirical analysis of the communication modules generated, providing quantitative performance data to support protocol selection. By measuring the memory footprint of each protocol within a real-world application, this work highlights the critical trade-offs between hardware-level and network-level communication approaches. These results serve as a valuable resource for system designers, enabling more informed data-driven decisions when selecting communication technologies for resource-constrained embedded systems.

Together, these contributions advance the state-of-the-art in model-driven engineering for distributed embedded systems. They not only extend the functionality of the IOPT-Tools framework but also establish a complete and validated workflow, supported by empirical evidence, that can guide both researchers and practitioners in the design of reliable and efficient distributed control solutions.

5.2 Limitations

Although the developed tool and the proposed workflow successfully meet the primary objectives of this dissertation, it is important to acknowledge several limitations of the current implementation. These limitations provide clear opportunities for future research and development.

The first limitation concerns the handling of multiple events. Currently, the API generates a complete and self-contained block of code for each individual event request. When multiple events are generated using the same communication protocol, the resulting output contains duplicated setup code and helper functions. This requires users to manually merge and de-duplicate the code, which is counter to the overall goal of full automation.

The second limitation relates to the scope of the protocol support. The tool was developed as a proof-of-concept and currently supports only three protocols: I²C, UART, and TCP/MQTT. Although these protocols cover a range of typical use cases, from local buses to network communication, other widely adopted industrial and embedded protocols, such as the Serial Peripheral Interface (SPI) and the Controller Area Network (CAN) bus, are not yet implemented.

A further limitation arises from the stateless design of the API. Each request is processed independently, which is effective in generating individual communication modules but prevents the implementation of a more complex project-based workflow. In its current form, the tool lacks the ability to manage a collection of events for a single

distributed system and to generate a consolidated and optimized output for all controllers involved.

Taken together, these limitations underscore the fact that while the current implementation demonstrates the feasibility and utility of automated communication code generation, it remains an early-stage solution. They highlight concrete directions for future work, particularly in extending protocol coverage, improving support for multievent systems, and evolving toward a more integrated and project-oriented workflow.

5.3 Future Work

The work presented in this dissertation establishes a solid foundation for the automated generation of communication modules. The limitations identified in the previous section provide a clear roadmap for future research and development that could further enhance the tool's capabilities and impact.

A first direction for future work is the development of a project-based generator. This would represent a significant evolution beyond the current stateless, per-event approach, enabling users to define an entire distributed system, including all its controllers and communication events. Such a generator would automatically manage dependencies, intelligently resolve code duplication, and produce a consolidated and optimized set of source files for the complete project. This would achieve complete automation and completely eliminate the need for manual code merging.

A second avenue concerns the expansion of protocol support. To increase the versatility and applicability of the tool in industrial and embedded domains, additional protocols must be implemented. Of particular relevance are the Serial Peripheral Interface (SPI), which enables high-speed local communication, and the Controller Area Network (CAN) bus, which is widely adopted in automotive and industrial automation systems. Inclusion of these protocols would significantly broaden the scope of the tool.

The third direction involves the creation of a more advanced user interface and integration mechanisms. On the one hand, a web-based Graphical User Interface (GUI) could be developed on top of the existing API, allowing users to configure communication links through intuitive forms and diagrams rather than manually constructing URLs. However, the current GET-based interface could be extended to a fully RESTful API with JSON-based data exchange. This would enable more robust programmatic integration with external development environments, including the IOPT-Tools platform itself, thereby enhancing usability and interoperability.

Together, these developments would strengthen the practical utility of the proposed tool, bringing it closer to a production-ready solution that supports a wide range of use cases and seamlessly integrates into existing model-driven workflows.

5.4 Final Remarks

The model-driven development paradigm offers a powerful approach to managing the complexity of modern embedded systems. By raising the level of abstraction, designers can focus on the core logic of a system while relying on automated tools for verification and implementation. This dissertation contributes to this paradigm by addressing a critical gap in the development of distributed systems with IOPT Petri nets. The automated tool presented here reduces development time, minimizes implementation errors, and empowers designers with empirical data to make better engineering decisions.This work serves as a valuable step toward more efficient and reliable methods for creating the next generation of distributed control systems.

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).

Full Generated Code for I²C Example

Listing A.1: I²C Example Code

```
1   // ================================================================
2   // Code for: buttonPress
3   // Generated by IOPT API for your 'DemoProject' project.
4   // All generated code is intended for the net_io file.
5   // Keep in mind if you use this protocol multiple times to implement different events you
          ↪ will get duplicated code, in that case add only what is diferent
6   //
7   // Code for Slave controller
8   // ================================================================
9
10  /*
11  * PART 1: Place this at the top of your file with the other global variables and includes.
12  */
13  #include <Wire.h>
14  #define I2C_SLAVE_ADDRESS_for_buttonPress 9 // Address for this device
15  #define CMD_TRIGGER_buttonPress 'P' // Command to activate the event
16  volatile bool buttonPress_trigger_flag = 0;
17  void receiveI2CEvent(int byteCount); // Function prototype
18
19
20  /*
21  * PART 2: Place this inside your 'DemoProject_InitializeIO()' function.
22  */
23  // Start the I2C bus as a slave with the defined address
24  Wire.begin(I2C_SLAVE_ADDRESS_for_buttonPress);
25  // Register the callback function that will be called when receiving data
26  Wire.onReceive(receiveI2CEvent);
27
28
29  /*
30  * PART 3: Place this inside your 'DemoProject_GetInputSignals()' function.
31  */
32  if (buttonPress_trigger_flag == 1) {
33  events->buttonPress = 1;
```

```
34  buttonPress_trigger_flag = 0; // Reset the flag to fire only once
35  } else {
36  events->buttonPress = 0; // Ensure the event is inactive by default
37  }
38
39
40  /*
41  * PART 4: Place this at the bottom of your file with the other helper functions.
42  */
43  void receiveI2CEvent(int byteCount) {
44  if (Wire.available() > 0) {
45  char command = Wire.read();
46  if (command == CMD_TRIGGER_buttonPress) {
47  buttonPress_trigger_flag = 1;
48  }
49  }
50  while (Wire.available() > 0) {
51  Wire.read();
52  }
53  }
54
55  // ================================================================
56  // Code for Master controller
57  // ================================================================
58
59
60  /*
61  * PART 1: Place this at the top of your file with the other global variables and includes.
62  */
63  #include <Wire.h>
64  #define I2C_SLAVE_ADDRESS_for_buttonPress 9 // Address for this device
65  #define CMD_TRIGGER_buttonPress 'P' // Command to activate the event
66  void triggerMasterSendSlaveEvent(byte slaveAddress, byte command);
67
68
69  /*
70  * PART 2: Place this inside your 'DemoProject_InitializeIO()' function.
71  */
72  while (!Serial); // Wait for Serial to be ready
73  Wire.begin(); // Start the I2C bus as a Master
74  Serial.println("I2C Master Initialized.");
75
76
77  /*
78  * PART 3: Place this inside your 'DemoProject_OutputSignals()' function.
79  */
80  if (events->buttonPress == 1) {
81  triggerMasterSendSlaveEvent(I2C_SLAVE_ADDRESS_for_buttonPress,CMD_TRIGGER_buttonPress);
82  }
83
```

```
84
85
86    /*
87    * PART 4: Place this at the bottom of your file with the other helper functions.
88    */
89    void triggerMasterSendSlaveEvent(byte slaveAddress, byte command) {
90    Serial.println("Attempting to send command to slave...");
91
92    // Step 1: Begin a transmission to the I2C slave device
93    Wire.beginTransmission(slaveAddress);
94
95    // Step 2: Send the command byte
96    Wire.write(command);
97
98    // Step 3: Stop the transmission and send the data
99    byte error = Wire.endTransmission();
100
101   // Check the status of the transmission
102   if (error == 0) {
103   Serial.println("Command sent successfully!");
104   } else {
105   Serial.print("Error sending command. Error code: ");
106   Serial.println(error);
107   Serial.println("Check connections and slave address.");
108   }
109   }
```

Full Generated Code for UART Example

Listing B.1: UART Example Code

```
1   // ================================================================
2   // Code for: UART Protocol
3   // Generated by IOPT API for your 'SystemA' project.
4   // All generated code is intended for the net_io file.
5   // Keep in mind if you use this protocol multiple times to implement
6   // different events you will get duplicated code, in that case add only what is different.
7   // ================================================================
8
9
10  // ================================================================
11  // Code for Receiver Controller (listens for event: 'toggleLED')
12  // ================================================================
13
14  /*
15  * PART 1: Place this at the top of your file with the other global variables and includes.
16  */
17  #include <HardwareSerial.h>
18
19  // -- UART Configuration --
20  HardwareSerial MySerial(2);
21  #define RXD2 16
22  #define TXD2 17
23
24  // -- Message and Flag Definitions --
25  const String message_uart_toggleLED = "trigger_toggleLED";
26  volatile bool toggleLED_trigger_flag = false; // flag only for receiving signals
27
28  // -- Function Prototypes --
29  void setupUart_toggleLED();
30  String receiveDataUart();
31  void waitMessageUart();
32
33  /*
34  * PART 2: Place this inside your 'SystemA_InitializeIO()' function.
```

```
35  */
36  setupUart_toggleLED();
37
38  /*
39  * PART 3: Place this inside your 'SystemA_GetInputSignals()' function.
40  */
41  waitMessageUart();
42  if (toggleLED_trigger_flag == true) {
43  events->toggleLED = 1;
44  toggleLED_trigger_flag = false; // Reset the flag to fire only once
45  } else {
46  events->toggleLED = 0; // Ensure the event is inactive by default
47  }
48
49  /*
50  * PART 4: Place this at the bottom of your file with the other helper functions.
51  */
52  void setupUart_toggleLED() {
53  MySerial.begin(9600, SERIAL_8N1, RXD2, TXD2);
54  Serial.println("UART communication initialized (Receiver).");
55  }
56
57  String receiveDataUart() {
58  if (MySerial.available()) {
59  String data = MySerial.readStringUntil('\n');
60  data.trim();
61  return data;
62  }
63  return "";
64  }
65
66  void waitMessageUart() {
67  String receivedMessage = receiveDataUart();
68  if (receivedMessage.length() > 0 && receivedMessage == message_uart_toggleLED) {
69  toggleLED_trigger_flag = true;
70  Serial.println("Correct message received. Flag for 'toggleLED' activated.");
71  }
72  }
73
74
75  // ================================================================
76  // Code for Sender Controller (sends event: 'toggleLED')
77  // ================================================================
78
79  /*
80  * PART 1: Place this at the top of your file with the other global variables and includes.
81  */
82  #include <HardwareSerial.h>
83
84  // -- UART Configuration --
```

```
85   HardwareSerial MySerial(2);
86   #define RXD2 16
87   #define TXD2 17
88
89   // -- Message Definitions --
90   const String message_uart_toggleLED = "trigger_toggleLED";
91
92   // -- Function Prototypes --
93   void setupUart_toggleLED();
94   void sendDataUart(String message);
95
96   /*
97   * PART 2: Place this inside your 'SystemA_InitializeIO()' function.
98   */
99   setupUart_toggleLED();
100
101  /*
102  * PART 3: Place this inside your 'SystemA_PutOutputSignals()' function.
103  */
104  if (events->toggleLED == 1) {
105  sendDataUart(message_uart_toggleLED);
106  events->toggleLED = 0; // IMPORTANT: Reset the event
107  }
108
109  /*
110  * PART 4: Place this at the bottom of your file with the other helper functions.
111  */
112  void setupUart_toggleLED() {
113  MySerial.begin(9600, SERIAL_8N1, RXD2, TXD2);
114  Serial.println("UART communication initialized (Sender).");
115  }
116
117  void sendDataUart(String message) {
118  MySerial.println(message);
119  Serial.print("Message sent: ");
120  Serial.println(message);
121  }
```

Full Generated Code for TCP/MQTT Example

Listing C.1: UART Example Code

```
1   // ================================================================
2   // Code for: TCP (MQTT) Protocol
3   // Generated by IOPT API for your 'SensorNetwork' project.
4   // All generated code is intended for the net_io file.
5   // Keep in mind if you use this protocol multiple times to implement different events you
        ↪ will get duplicated code, in that case add only what is diferent
6   // ================================================================
7
8   // ================================================================
9   // Receiver Controller (listens for event: 'alert')
10  // ================================================================
11
12  /*
13  * PART 1: Place at the top with other globals
14  */
15  #include <WiFi.h>
16  #include <PubSubClient.h>
17
18  const char* ssid = "yourNetworkName";
19  const char* password = "yourNetworkPassword";
20
21  const char* mqtt_broker = "broker.hivemq.com";
22  const int mqtt_port = 1883;
23  const char* client_id = "ESP32_IOPT";
24  const char* topic_sub_alert = "sensors/events";
25
26  const char* message_tcp_alert = "trigger_alert";
27  volatile bool alert_trigger_flag = false;
28
29  WiFiClient espClient;
30  PubSubClient client(espClient);
31
```

```
32  void tcpMqttInitializeIO();
33  void reconnect(const char* topic);
34  void loopDelayTcp(const char* topic);
35  void callback(char* topic, byte* payload, unsigned int length);
36
37  /*
38  * PART 2: Place inside SensorNetwork_InitializeIO()
39  */
40  tcpMqttInitializeIO();
41
42  /*
43  * PART 3: Place inside SensorNetwork_GetInputSignals()
44  */
45  if (alert_trigger_flag) {
46  events->alert = 1;
47  alert_trigger_flag = false;
48  } else {
49  events->alert = 0;
50  }
51
52  /*
53  * PART 4: Place inside SensorNetwork_LoopDelay()
54  */
55  loopDelayTcp(topic_sub_alert);
56
57
58  /*
59  * PART 5: Helper functions (place at the bottom)
60  */
61  void tcpMqttInitializeIO() {
62  Serial.begin(115200);
63  WiFi.begin(ssid, password);
64  Serial.print("Connecting to WiFi...");
65  while (WiFi.status() != WL_CONNECTED) {
66  delay(500);
67  Serial.print(".");
68  }
69  Serial.println("\nWiFi connected!");
70  client.setServer(mqtt_broker, mqtt_port);
71  client.setCallback(callback);
72  }
73
74  void reconnect(const char* topic) {
75  while (!client.connected()) {
76  Serial.print("Connecting to MQTT broker...");
77  if (client.connect(client_id)) {
78  Serial.println("connected!");
79  client.subscribe(topic);
80  } else {
81  Serial.print("failed, rc=");
```

```
82    Serial.println(client.state());
83    delay(5000);
84    }
85    }
86    }
87
88    void loopDelayTcp(const char* topic) {
89    if (!client.connected()) {
90    reconnect(topic);
91    }
92    client.loop();
93    }
94
95    void callback(char* topic, byte* payload, unsigned int length) {
96    String msg = "";
97    for (int i=0; i<length; i++) msg += (char)payload[i];
98    msg.trim();
99    if (msg == String(message_tcp_alert)) {
100   alert_trigger_flag = true;
101   Serial.println("Correct TCP message received. Flag activated for 'alert'.");
102   }
103   }
104
105
106   // =================================================================
107   // Sender Controller (sends event: 'alert')
108   // =================================================================
109
110   /*
111   * PART 1: Place at the top with other globals
112   */
113   #include <WiFi.h>
114   #include <PubSubClient.h>
115
116   const char* ssid_s = "yourNetworkName";
117   const char* password_s = "yourNetworkPassword";
118
119   const char* mqtt_broker_s = "broker.hivemq.com";
120   const int mqtt_port_s = 1883;
121   const char* client_id_s = "ESP32_IOPTSender";
122   const char* topic_pub_alert = "sensors/events";
123
124   const char* message_tcp_alert = "trigger_alert";
125
126   WiFiClient espClient_s;
127   PubSubClient client_s(espClient_s);
128
129   void tcpMqttInitializeIO_Sender();
130   void reconnectSender(const char* topic);
131   void loopDelayTcpSender(const char* topic);
```

```
132  void sendMessageTcp(const char* topic, const char* message);
133
134  /*
135  * PART 2: Place inside SensorNetwork_InitializeIO()
136  */
137  tcpMqttInitializeIO_Sender();
138
139  /*
140  * PART 3: Place inside SensorNetwork_PutOutputSignals()
141  */
142  if (events->alert == 1) {
143  sendMessageTcp(topic_pub_alert, message_tcp_alert);
144  events->alert = 0;
145  }
146
147  /*
148  * PART 4: Place inside SensorNetwork_LoopDelay()
149  */
150  loopDelayTcp(topic_pub_alert);
151
152  /*
153  * PART 5: Helper functions (place at the bottom)
154  */
155  void tcpMqttInitializeIO_Sender() {
156  Serial.begin(115200);
157  WiFi.begin(ssid_s, password_s);
158  Serial.print("Connecting to WiFi...");
159  while (WiFi.status() != WL_CONNECTED) {
160  delay(500);
161  Serial.print(".");
162  }
163  Serial.println("\nWiFi connected!");
164  client_s.setServer(mqtt_broker_s, mqtt_port_s);
165  }
166
167  void reconnectSender(const char* topic) {
168  while (!client_s.connected()) {
169  Serial.print("Connecting to MQTT broker...");
170  if (client_s.connect(client_id_s)) {
171  Serial.println("connected!");
172  } else {
173  Serial.print("failed, rc=");
174  Serial.println(client_s.state());
175  delay(5000);
176  }
177  }
178  }
179
180  void loopDelayTcpSender(const char* topic) {
181  if (!client_s.connected()) {
```

54

```
182  reconnectSender(topic);
183  }
184  client_s.loop();
185  }
186
187  void sendMessageTcp(const char* topic, const char* message) {
188  Serial.print("Event detected! Publishing to topic: ");
189  Serial.println(topic);
190  client_s.publish(topic, message);
191  }
```