NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

DUARTE JOSÉ RIBEIRO TAVARES

BSc in Electrical and Computer Engineering Sciences

COMMUNICATION MODULES FOR DISTRIBUTED CONTROLLERS SPECIFIED THROUGH IOPT MODELS

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

COMMUNICATION MODULES FOR DISTRIBUTED CONTROLLERS SPECIFIED THROUGH IOPT MODELS

DUARTE JOSÉ RIBEIRO TAVARES

BSc in Electrical and Computer Engineering Sciences

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

Examination Committee

Chair: Nuno Filipe Silva Veríssimo Paulino
Associate Professor with Habilitation, NOVA University Lisbon

Rapporteur: Filipe de Carvalho Moutinho
Assistant Professor, NOVA University Lisbon

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

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2025

Communication Modules for Distributed Controllers Specified through IOPT Models

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.3.3) [27].

Acknowledgements

I would like to express my sincere gratitude to all those who have supported me throughout my Master's journey. This dissertation would not have been possible without their guidance, encouragement, and patience.

First and foremost, I extend my deepest gratitude to my advisor, **Professor Luís Filipe dos Santos Gomes**. I am sincerely thankful for the opportunity and for his constant availability to guide me through this entire process. A special thanks for the challenge he proposed, which allowed me to deepen my knowledge in the fields of digital systems and telecommunications, areas I am passionate about and in which I currently work. His guidance was fundamental to this project.

On a personal note, I could not have reached this point without the support of my family and friends.

To my family: thank you for making it possible for me to study and for being there during the most difficult moments. Your support gave me strength when I needed it the most.

Finally, a very special thanks to my girlfriend, **Cátia**. Thank you for always believing in me, for motivating me to push forward, and for reminding me not to give up. Your presence was essential throughout this journey.

To you all, thank you.

Abstract

Model-driven development, particularly using formalisms like Input-Output Place-Transition (IOPT) Petri nets, offers a robust approach for designing complex embedded controllers. The IOPT-Tools framework supports this by automating the generation of controller logic, but a significant gap exists in its ability to automatically generate the communication infrastructure required for distributed systems. This necessitates a manual, time-consuming, and error-prone process for implementing inter-controller communication links.

This dissertation addresses this gap by presenting the design, implementation, and validation of a web-based Application Programming Interface (API) for the automated generation of communication modules. The tool dynamically produces C++ code for three distinct protocolscommunication technologies, I2C, UART, and TCP/MQTT, designed for direct integration into the IOPT-Tools workflow.

The utility and performance of the generated code were validated through a comprehensivepractical case study of a three-conveyor distributed system managed by four controllers.. An empirical analysis yielded quantitative performance metrics, demonstrating the significant trade-offs between protocols: hardware-level protocols such I^2C (185 µs latency, negligible memory overhead) offered superior real-time performance, whereas the network-level TCP/MQTT (45 ms latency, >26 kB overhead) provided greater functional flexibility. The dynamic behavior of the implemented system was also verified against the original IOPT models using simulator-generated timing diagramstiming diagrams generated by the IOPT-Tools simulator..

The results confirm that the developed tool successfully accelerates distributed system developmentsignificantly reduces implementation time by automating the creation of the communication infrastructure , reduces implementation errors, and empowers designers with the empirical data needed to make informed, data-driven decisions when balancing performance requirements against the resource constraints of embedded systems.

Resumo

O desenvolvimento orientado a modelos, particularmente com o uso de formalismos como as Redes de Petri Input-Output Place-Transition (IOPT), oferece uma abordagem robusta para o projeto de controladores embebidos complexos. A plataforma IOPT-Tools suporta este paradigma, automatizando a geração da lógica dos controladores, mas existe uma lacuna significativa na sua capacidade de gerar automaticamente a infraestrutura de comunicação necessária para sistemas distribuídos. Isto exige um processo manual, demorado e propenso a erros para implementar as ligações de comunicação entre controladores.

Esta dissertação aborda esta lacuna, apresentando o desenho, a implementação e a validação de uma Application Programming Interface (API) baseada na web para a geração automatizada de módulos de comunicação. A ferramenta produz dinamicamente código C++ para três protocolos distintostecnologias de comunicação distintas, I²C, UART e TCP/MQTT, projetado para integração direta no fluxo de trabalho das IOPT-Tools.

A utilidade e o desempenho do código gerado foram validados através de um estudo de caso abrangenteprático de um sistema distribuído de quatro controladores para três tapetes rolantes. Uma análise empírica forneceu métricas de desempenho quantitativas, demonstrando os significativos compromissos (*trade-offs*) entre os protocolos: protocolos de hardware como o I²C (latência de 185 µs, consumo de memória residual) ofereceram um desempenho superior em tempo real, enquanto o protocolo de rede TCP/MQTT (latência de 45 ms, consumo superior a 26 kB) proporcionou maior flexibilidade funcional. O comportamento dinâmico do sistema implementado foi também verificado através de diagramas temporais gerados pelo simulador do IOPT-Tools.

Os resultados confirmam que a ferramenta desenvolvida acelera com sucesso o desenvolvimentoreduz significativamente o tempo de implementação ao automatizar a criação da infraestrutura de comunicação de sistemas distribuídos, reduz os erros de implementação e capacita os projetistas com os dados empíricos necessários para tomar decisões informadas e baseadas em dados, ao equilibrar os requisitos de desempenho com as restrições de recursos dos sistemas embebidos.

. T

Contents

List of Figures

Listings

Acronyms

AC Asynchronous Channel *(pp. 10, 11, 25)*
AJAX Asynchronous JavaScript and XML *(p. 20)*
API Application Programming Interface *(pp. 3, 4, 22–28, 34, 35, 37, 40, 42–44, 47, 51–53)*

FIFO First-In-First-Out *(pp. 5, 14, 18)*
FPGA Field-Programmable Gate Array *(pp. 17, 21)*

GALS Globally Asynchronous, Locally Synchronous *(pp. 2, 3, 5, 8, 10–16, 18, 19, 22, 25, 26)*
GUI Graphical User Interface *(pp. 21, 53)*

I/O Input/Output *(pp. 7–9, 21)*
I²C Inter-Integrated Circuit *(pp. vi, viii, 3, 5, 22, 23, 25–28, 30, 35–37, 40–42, 46, 50–52)*
IEEE Institute of Electrical and Electronics Engineers *(p. 16)*
IOPT Input-Output Place-Transition *(pp. 2, 3, 5–8, 10–27, 36–38, 40–42, 44, 47, 50–52, 54)*
ISR Interrupt Service Routine *(pp. 28, 30, 32, 36)*

MQTT Message Queuing Telemetry Transport *(pp. viii, 3, 15, 16, 22, 23, 25, 27, 28, 32, 34–37, 41, 44, 46, 48–52)*

NoC Network-on-Chip *(p. 18)*

PCB Printed Circuit Board *(p. 13)*
PN Petri Net *(p. 6)*
PNML Petri Net Markup Language *(pp. 8, 20)*

RX Receive *(pp. 13, 27, 36)*

SPI Serial Peripheral Interface *(pp. 3, 5, 12, 13, 18, 53)*
SVG Scalable Vector Graphics *(p. 20)*

TCP Transmission Control Protocol *(pp. viii, 3, 15, 16, 22, 23, 27, 35, 37, 41, 44, 46, 48–52)*

TD Time Domain *(pp. 10, 11)*

TX Transmit *(pp. 13, 27, 36)*

UART Universal Asynchronous Receiver-Transmitter *(pp. viii, 3, 5, 13, 14, 18, 22, 23, 25, 27, 30, 35, 37, 41–43, 46–52)*

UDP User Datagram Protocol *(pp. 15, 16)*

VHDL VHSIC Hardware Description Language *(pp. 19, 21)*

XML Extensible Markup Language *(p. 20)*

1 Introduction

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

1.1 Context and Motivation of the Work

The contemporary engineering landscape is undergoing a profound transformation, often termed the Fourth Industrial Revolution or Industry 4.0. This paradigm is characterized by the deep integration of cyber-physical systems (CPS), the Internet of Things (IoT), and cloud computing into industrial processes [23]. At the core of this evolution lies a fundamental shift from centralized monolithic control architectures towards highly distributed and interconnected embedded systems [26]. These systems are composed of numerous intelligent nodes, sensors, actuators, and controllers, that must cooperate in a coordinated manner to achieve complex tasks, from smart manufacturing to autonomous logistics.

This distribution offers significant advantages, such as improved modularity, scalability, and fault tolerance. However, it also introduces unprecedentedincreased levels of complexity in system design, verification, and real-time coordination [11]. Managing the concurrency, synchronization, and communication inherent in these systems requiresbenefits significantly from formal modeling techniques capable of capturing their dynamic behavior with mathematical rigor. It is within this modern context that Petri nets, a formalism with a long-standing history, find renewed and critical relevance.

The concept of Petri nets was first introduced by Carl Adam Petri in his 1962 dissertation "Kommunikation mit Automaten" [33]. Originally proposed as a modeling technique for distributed and concurrent systems in computer science, Petri nets have since become widely adoptedfound significant application in the fields of embedded systems and software development.

Petri nets provide a robust and intuitive framework for 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 [30].

Among the numerous tools available for Petri net modeling, the Input-Output Place-Transition (IOPT)-Tools environment underpins a specialized approach to developing controllers through IOPT models, a specific class of Petri nets. The IOPT-Tools framework, accessible at http://gres.uninova.pt/IOPTTools/`http://gres.uninova.pt/IOPT-Tools-V1.2/login.php`, 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 Globally Asynchronous, Locally Synchronous (GALS) paradigm, has intensified the need for efficient and dependable communication between these distributed sub-models. As systems become more modular and geographically distributed, ensuring low-latency data exchange and maintaining operational consistency pose significant challenges. These challenges are particularly acute in the context of distributed IOPTsub-models and GALS, as manual communication handling can lead to extensive development effort, introduce the 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. This absence of automation relegates the implementation of inter-controller communication to a manual, ad-hoc process. Such a process is not merely inefficient; it is fraught with significant risks that can undermine the reliability and economic viability of the system. Manual coding of communication protocols is inherently error-prone, increasing the likelihood of subtle implementation defects, such as race conditions and deadlocks. which are notoriously difficult to detect and debug in a distributed environmentThese issues are particularly difficult to detect and debug within the complex landscape of Cyber-Physical Systems [6]. Furthermore, this approach elevates development costs and extends the time-to-market, which are critical concerns in competitive industrial settings [42]. It also places a heavy cognitive burden on system designers, requiring them to possess deep expertise in both high-level control modeling and the low-level intricacies of multiple hardware and network protocols, which complicates system maintenance and hinders the reusability of the resulting code [41].

The main problem is to establish a robust mechanism that enables the distributed controllers, each executing an independent IOPTsub-model, to exchange data efficientlyThe

main problem is to establish a systematic mechanism that enables the distributed controllers, each executing an independent IOPT sub-model, to exchange data reliably and with optimized resource usage. 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 structured around two main objectives. The first is a comparative analysisthe evaluation of communication technologies, encompassing both wired point-to-point networks such as I^2C, Serial Peripheral Interface (SPI), and UART and wireless solutions including Wi-Fi and Bluetooth, to determine their suitability for the proposed tool. These technologies are evaluated against criteria such as simplicity, reliability, and latency in order to identify their suitability for distributed embedded systems. The second objective is the development of an automated code generation tool capable of producing the communication modules required for each IOPT sub-model. By automating this process, the tool ensures efficient data exchange between controllers while maintaining robustness and responsiveness within the overall system.

The successful fulfillment of these objectives leads to four primary contributions. First, a functional web-based Application Programming Interface (API) was designed and implemented to automate the generation of communication modules for three widely used protocols (I^2C, UART, and TCP/MQTT), directly addressingbridging the communication gap in the IOPT-Tools framework. Second, the research validates an end-to-end model-driven workflow, spanning from high-level IOPT Petri net specifications to a deployed multi-controller hardware implementation, as demonstrated through a comprehensive real-world case study. Third, an empirical performance analysis of the generated modules quantifies memory footprint and communication latency, providing a data-driven basis for selecting communication technologies in resource-constrained systems. Finally, the developed API has been publicly released as an open-source artifact, thereby promoting transparency, enabling reproducibility, and offering a foundation for future community-driven extensions. To further support reproducibility and long-term development, the software is available in a dedicated source code repository.

1.4 Dissertation Structure

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

- **Chapter 1 - Introduction:** This chapter introduces the context and motivation for the research, defines the problem statement, presents the main objectives, and outlines

the structure of the document.

- **Chapter 2 - State of the Art:** This chapter provides a thorough review of the theoretical and technological foundations for the work, covering Petri nets, the IOPTformalism, GALS architectures, and a detailed analysis of relevant communication protocols.
- **Chapter 3 - System Design and Implementation:** This chapter details the architecture and design of the automated code generation tool. It presents the rationale for the design, the mapping from model constructs to code, the formal API specification, an analysis of the generated code, and the methodology for the tool's validation.
- **Chapter 4 - Case Study and Performance Analysis:** This chapter presents a comprehensive case study applying the developed tool to a real-world multi-controller automation system. Details the implementation process and provides an empirical performance analysis of the generated communication modules, focusing on memory footprint and dynamic behavior.
- **Chapter 5 - Conclusion:** The final chapter summarizes the dissertation, reiterates the main contributions, discusses the limitations of the current work, and proposes concrete directions for future research.

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 communication technologies, highlighting their significance in enabling efficient and reliable communication within distributed systems. The discussion then examines various communication protocols, including I^2C, SPI, UART, RS-485 and First-In-First-Out (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 [33]. In subsequent decades, the theoretical foundations of Petri nets were strengthened by seminal results on decision problems such as reachability and liveness [30]. Additionally, the establishment of the annual International Conference on Applications and Theory of Petri Nets and Concurrency provided a forum to advance both theory and practice [12], firmly establishing Petri nets as a formal graphical language for discrete-event and concurrent systems modeling [48].

the theoretical foundations of Petri nets were strengthened by seminal results on decision problems such as reachability and liveness [30], while the annual International Conference on Applications and Theory of Petri Nets and Concurrency provided a forum to advance both theory and practice [12], firmly establishing Petri nets as a formal graphical language for discrete-event and concurrent systems modeling [48].

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 [30]. 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 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 [39]. 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 [39].

Graphically, P/T nets serve as intuitive visual-communication aids for stakeholders, such as clients, manufacturers and users, supporting model comprehension and system specification [38].

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 [30]. This inherent complexity, where Petri-net-based models can become too large for analysis even for a modest-size system, is mitigated in the IOPT-Tools environment through its support for hierarchical modeling and decomposition into independent sub-models, which aids in managing the complexity for distributed systems.

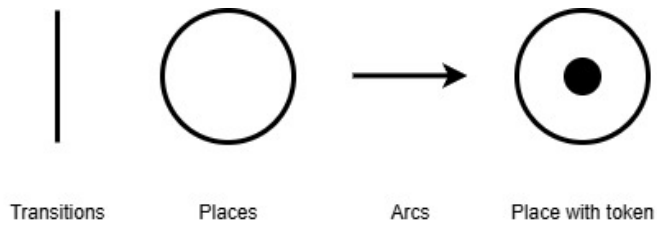

Figure 2.1: Petri net elements.

The formal definition of a P/T net states that a Petri net $PetriNet(PN)$ is defined as the tuple (meaning they cannot be modified once created) $PN = (P, T, F, W, M_0)$ [30], 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_4 to continue the cycle.

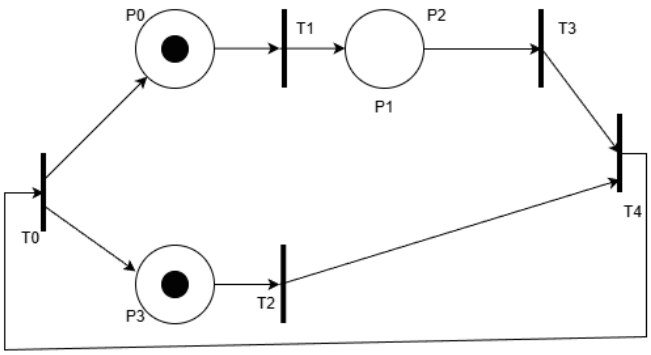

Figure 2.2: Example of Place/Transition net.

2.2 *Input-Output Place-Transition* Petri Nets

The **IOPT Petri Nets** were created to model controllers interacting with external environments through the use of (Input/Output (I/O)) interface[18]. They are considered *non-autonomous* where non-autonomous means that external signals can enable or disable transitions [32]. 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 system[18].

The IOPT frameworknet interfaces 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 [18]. 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 [3] and [31]. The introduction of prioritization allows for effective conflict resolution among transitions, while test arcs facilitate the implementation of fair arbitration mechanisms [13].

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 (Petri Net Markup Language (PNML)) and extends it with Ecore-based representations to capture I/O, timing, and communication aspects [18].

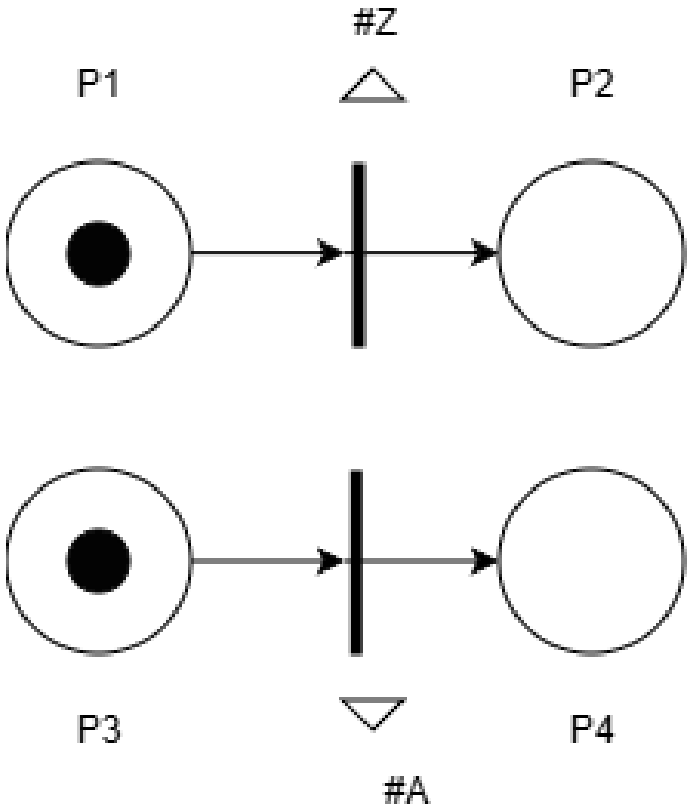

Figure 2.3: Example of IOPT Petri net.

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.Figure 2.3 depicts two distinct Petri nets where transitions are associated with specific communication actions: transition T_1 involves the output action $> \#Z$, and transition T_2 involves the input action $\#A >$. This setup illustrates the mechanism by which locally asynchronous components can achieve synchronization. Structurally, the

sub-nets function independently, and while T_1 and T_2 are enabled by their local markings, their firing is constrained by the requirement for a simultaneous complementary event in the environment. In the specific state shown, synchronization cannot occur because the actions do not match ($Z \neq A$); consequently, each transition must wait for a corresponding partner to fire.

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 [10].causal relationships between Input/Output (I/O) signals, ensuring that the distributed system maintains the same behavioral semantics as the original model [10].

- **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 [44], 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) signalsensuring the correct functional integration of the independent modules [10].

- **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 [8]. The use of net splitting creates smaller and more manageable systems, supporting modular design approaches and allowing for incremental development and analysis [17].

  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 [7]:

  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 [7], [8]. When a node is split, the original and its copies are linked in a "synchrony set" or "synchrony group" [8]. One transition in the set is designated as the "master" and the others as "slaves" [7], [8]. Semantically, all transitions in a synchrony set are intended to fire simultaneously, as if they were a single fused transition [7]. This ensures that, at the model level, the set of sub-models is behaviorally equivalent or similar to the original, monolithic net [7], [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 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 [5].

An extension of GALS systems applied to IOPT models is presented in [28], where the author introduces an attribute that specifies the Time Domain (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 Channel (AC). 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:

$$IOPT_{2GALS} = (IOPT, ACs, TDs) \tag{2.1}$$

Where:

- **IOPT** is the tuple defined by:

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

- **ACs** is the set of Asynchronous Channels, representing directed connections between a master transition (t_m) and a slave transition (t_s):

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

  Note: An AC always connects two transitions with different Time Domains.

- **TDs** is the set of time domains covering places, transitions, and channels:

$$TDs = TDs_p \cup TDs_t \cup TDs_{ac} \tag{2.4}$$

The integration of GALS into IOPT Petri nets involvescan be systematically achieved by decomposing a global system model into locally synchronous modules that communicate asynchronously. This decomposition creates TD and AC, facilitates modular design, enables each module to operate at its own pace without the need for global synchronization, thus improving scalability and flexibility [28].

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.

2.4 Communication support technologies

In this section, various communication technologies utilised within (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,

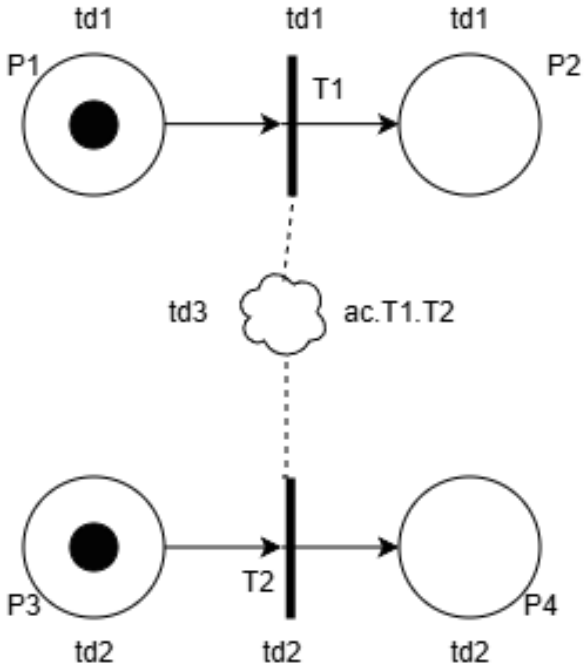

Figure 2.4: Example of Petri net with GALS system.

the chapter aims to equip readers with the knowledge necessary to navigate the complexities of modern integrated circuit development and to harness the benefits of GALS architectures for optimised system performance and reliability.

2.4.1 I^2C

The I^2C (Inter-Integrated Circuit) protocol is described as a multi-master, serial, and single-ended bus that requires only two lines for communication: the clock line (SCL) for synchronization and the data line (SDA) for transmitting information. This simplicity is underscored by the minimal hardware requirements, just three connections (SCL, SDA, and GND), making it an ideal choice for connecting the central controller (master) with multiple local controllers (slaves) [34].

Despite its advantages, the I^2C protocol presents certain limitations that may hinder its suitability for high-performance or large-scale applications. The standard I^2C bus supports data rates of up to 3.4 Mbps in high-speed mode, which may prove inadequate for systems that demand greater throughput. Moreover, the inherent bus capacitance imposes constraints on the number of connected deviceswhile the protocol's 7-bit addressing theoretically supports up to 127 devices, the inherent bus capacitance imposes practical constraints on the actual number of connected nodes, the permissible bus length and may have problems with bus contention, thereby limiting the scalability of the system [47].

Since I^2C assumes that both devices operate within the same clock domain, it is not suitable for the asynchronous components of GALS 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. Although I^2C devices rely on a shared clock signal (SCL) for data synchronization during transmission, the controllers themselves often operate in independent clock domains. This characteristic makes I^2C a viable option for communication within the locally synchronous parts of IOPT sub-models, where minimal wiring is a priority. Furthermore, when employed for inter-controller communication in GALS systems, it requires specific mechanisms to ensure proper clock domain crossing, effectively bridging the synchronous bus transfer with the asynchronous coordination logic.

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 [21]. 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 [21].

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 (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 [24].

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 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. Like I^2C, SPI employs a synchronous clock signal (SCK) generated by the master to synchronize data transmission. While this ensures strict timing during the transaction, it does not imply that the connected devices must operate within the same system clock domain. Consequently, SPI remains a valid and efficient option for linking distinct asynchronous domains within a GALS system, provided that appropriate mechanisms, such as buffering or interrupts, are implemented to handle the asynchronous nature of message arrival 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 [49]. 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 [37].

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 [37].

While UART shares the fundamental mechanism of parallel-to-serial conversion with other protocols like I^2C and SPI, its distinguishing feature is its fully asynchronous operation. Instead of a shared clock signal, it relies on strict timing based on a pre-agreed baud rate and specific framing bits (start and stop) to synchronize data transfer between the transmitter and receiver. 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 [37].

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 [20]. 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 [46]. 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 [4]. 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.

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 Array (FPGA)s or hardware-in-the-loop systems, often for crossing clock domains [4]. While this makes FIFO + Handshake a fundamental building block for robust asynchronous communication in theoretical GALS models or internal chip interconnects, its requirement for parallel wiring often makes it less practical for physical inter-controller communication compared to serial protocols.

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 [45].

At the destination, received packets undergo processing according to the specific transport protocol integrated with IP. The most frequently employed transport protocols include Transmission Control Protocol (TCP) and User Datagram Protocol (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 [35] and taught widely in foundational networking texts [25]. It establishes a connection via a three-way handshake before data transmission begins, a process detailed exhaustively in technical literature [43]. To ensure reliability, TCP uses mechanisms such as sequence numbers, acknowledgments, and retransmission of lost packets [35]. 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 [25, 43]. 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 [36]. UDP is considered an unreliable protocol as it does not guarantee delivery, order, or duplicate protection [25]. 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 [9]. 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.

Beyond transport protocols, higher-level messaging protocols have been developed to

facilitate lightweight asynchronous communication between distributed systems. Two particularly relevant protocols in the context of distributed IOPT sub-models and GALS-based communication are **MQTT** (Message Queuing Telemetry Transport) and its derivative **MQTT-SN** (MQTTfor Sensor Networks).

MQTT is a lightweight publish/subscribe messaging protocol originally designed for machine-to-machine (M2M) communication over unreliable or constrained networks [29, 2]. Typically running on top of TCP, MQTT employs a broker-based architecture in which clients publish messages to topics managed by a central broker, while subscribers receive messages corresponding to the topics of interest. This decoupling of producers and consumers improves scalability and simplifies asynchronous communication in distributed environments.

One of MQTT's defining features is its three Quality of Service (QoS) levels, which allow tailoring reliability guarantees to application requirements: QoS 0 (at most once), QoS 1 (at least once), and QoS 2 (exactly once) [29]. These options provide flexibility between minimizing latency and ensuring reliable delivery. In the context of this implementation, QoS 0 was selected to prioritize low latency and minimize network overhead, ensuring that the communication module remains lightweight for the embedded controllers. The low overhead and efficient packet structure make MQTT particularly suitable for distributed IOPT systemsWhile exhibiting higher latency than local hardware protocols like I^2C, the low overhead and efficient packet structure make MQTT particularly suitable for distributed IOPT system, where GALS modules require asynchronous, topic-driven coordination with configurable trade-offs between latency and reliability.

MQTT-SN extends the MQTT concepts to environments where devices may have severe resource constraints or lack complete TCP / IP support, such as sensor networks and embedded systems [40, 50]. Unlike MQTT, which assumes a TCP transport, MQTT-SN is designed to run over connectionless transports like UDP or even directly over serial or Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 links. Introduces optimizations such as replacing string-based topic names with short integer identifiers and providing lightweight session management [40].

These adaptations significantly reduce message overhead, making MQTT-SN more appropriate for constrained distributed IOPT sub-models deployed in resource-limited contexts. Moreover, the combination of UDP-based transport and compact message formats reduces communication latency, an essential requirement in GALS systems where asynchronous interactions between modules must be timely. Reliability, when needed, can be addressed through MQTT-SN's QoS support or modeled explicitly at the IOPT Petri net layer, ensuring that protocol simplicity does not compromise system correctness.

Although MQTT is widely adopted in IoT applications due to its reliability and the structured publish / subscribe model, its reliance on TCP introduces connection management overhead and potential latency [2]. MQTT-SN, on the contrary, trades some of the reliability guarantees of TCP for lower latency and efficiency, which makes it particularly attractive for asynchronous real-time communication in distributed IOPT/GALS systems.

The choice between them thus depends on the constraints of the system: MQTT may be more suitable when strong reliability and established broker infrastructure are required, whereas MQTT-SN aligns better with scenarios that emphasize lightweight, low-latency exchanges under strict resource constraints.

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

The development of distributed control systems, under the (GALS) paradigm [28, 5], 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.

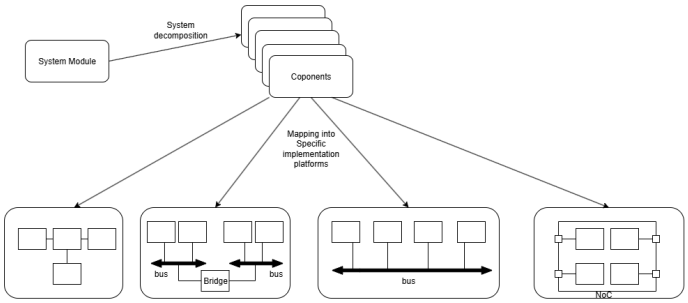

Figure 2.5: From models to code through model partitioning and mapping.

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 [18, 19]. 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) [17, 14]. 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 (FPGA), or even distinct software processes. As illustrated in Figure 2.5, the interaction between these distributed components must be realized through a defined network structure. The choice of network topology is essential and can have different variations, including:
  - **Direct Connections:** Point-to-point links between two components, often suitable for dedicated, high-throughput, or low-latency communication.
  - **Shared Buses:** Several components share a common communication channel, such as a "bus" or a "bridge"-connected "multi-bus" system (Figure 2.5), which necessitates mechanisms (e.g., communication protocols) for arbitration and addressing.
  - **Network-on-Chip (NoC) Structures** **Network-Level Communitation Protocols** 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 [28]. 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 [5]. 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 [28, 18].

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 [17].

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 [49, 37] 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 [34, 47], or SPI, where multiple slaves can be managed via individual select lines [21, 24].

- Networks-on-Chip (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 modeled within IOPT nets extended for GALS architectures 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 [18]. As previously mentioned [18], 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 VHSIC Hardware Description Language (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 [16]. This feature streamlines the transition from design to deployment, allowing for efficient and error-free implementation of the controller in either software or hardware [22].

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 [14]. This editor leverages Asynchronous JavaScript and XML (AJAX) principles, dynamically manipulating PNML (Petri Net Markup Language) data in an Extensible Markup Language (XML) DOM document and utilizing XSL transformations to generate real-time Scalable Vector Graphics (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 [14]. 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 [14].

For system verification and debugging, IOPT-Tools incorporates a robustcomprehensive **verification engine** primarily composed of a state-space generator and a query system [22, 32]. 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 [16]. 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 [32].

The framework also includes a **simulator tool** for executing and debugging IOPT models within the web browser [32]. 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 [32]. The simulator employs a compilation execution strategy, dynamically generating JavaScript code for model semantics to ensure high performance [32].

Regarding **code generation**, IOPT-Tools offers automatic tools to produce C and VHDL from IOPT models [22, 16]. 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 [22]. 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 [16]. These generated components are synchronous, requiring input signals to be synchronized with a clock [16]. 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 [16]. 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), Graphical User Interface (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) [22]. 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 [32, 22].The IOPT-Tools framework encompasses a suite of specialized and interconnected tools that support specific stages of the development lifecycle. Key components include **Split** (integrated for model decomposition into sub-models, as discussed in Section 2.2.1), **HIPPO** (employed for structural Petri net analysis), and **SnoopyIOPT** (an alternative editor). Additionally, the environment features **Animator** (for animated synoptics), the **GUI Generator** for FPGA, and the **Configurator** (for I/O pin assignment). The framework's continuous evolution is further evidenced by features such as the waveform viewer within the simulator and ongoing research into in-circuit emulation [32, 22].

2.7 Chapter Summary

This chapter established the theoretical and technological foundations necessary for the development of the proposed code generation tool. It began by reviewing the formalism of Petri nets and their IOPT extension, highlighting their suitability for modeling embedded controllers and the importance of net decomposition in the design of distributed systems. The GALS paradigm was introduced as the architectural framework for these systems, emphasizing the distinction between local synchronous execution and global asynchronous communication.

A review of communication technologies, ranging from hardware-level protocols like I^2C and UART to network-level protocols like TCP/MQTT, demonstrated the diverse options available for implementing the asynchronous channels required by GALS models. Finally, an analysis of the IOPT-Tools environment revealed a critical gap: while the platform excels in modeling and logic verification, it lacks automated support for generating the inter-controller communication infrastructure. This limitation, which currently forces error-prone manual implementation, defines the central problem addressed in this dissertation and motivates the design of the automated solution presented in the following chapter.

3

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 IOPT Petri nets [18, 3, 19]. However, a significant limitation arises in distributed control systems, particularly under the GALS paradigm, where decomposed sub-models require intercommunication [28, 17]. 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-modelsprocessing the communication specifications of 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.This chapter presents the design, implementation, and functional verification of the proposed tool. It begins with an overview of the system architecture, developed as a web-based Application Programming Interface (API) designed for integration 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: I^2C, 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 System Architecture

The tool is implemented as a server-side web Application Programming Interface (API), an architectural choice that decouples the code generation logic from the end-user's local environment while providing a platform-independent and highly accessible solution. At the core of the system lies a single PHP script, `api.php`, hosted on a web server, which is responsible for processing all incoming requests[1]. The overall architectural workflow, depicted in Figure 3.1, illustrates the sequence of interactions from the user's request to the delivery of the generated C++ code.

The process begins with the user, or an automated client, constructing a Hypertext Transfer Protocol (HTTP) GET request containing a set of URL parameters. These parameters specify the desired communication protocol (I^2C, UART, or TCP/MQTT) and its configuration, including details such as slave addresses, baud rates, or topic names, as defined in Section 3.4. Upon receiving the request, the web server, which can be implemented using common platforms such as Apache or Nginx, acts as the public-facing entry point and forwards the request to the `api.php` script for processing.

Within the PHP script, the input parameters are first parsed and sanitizedfiltered, then validated against the API specificationpredefined constraints to ensure completeness and correctness. Based on the selected protocol, the script retrieves the corresponding pre-written C++ code template, which contains placeholders for the protocol-specific configuration values. These templates serve as skeletons for each supported communication protocol, enabling modularity and separation of generation logic from code structurethe decoupling of the processing logic from the static code templates. The script dynamically populates the template with the user-provided parameters and assembles the final C++ source code as a string, ready for integration.module, formatted for direct inclusion into the controller's firmware. In cases of missing or invalid parameters, the script generates an HTTP 400 error response with a descriptive message, ensuring robust error handling.

Finally, the generated C++ code is returned to the user within the body of the HTTP response. This output is formattedstructured as a set of functions and definitions specifically designed for direct integration intoinclusion within the `net_io.cpp` file produced by the IOPT-Tools environment. By centralizing the code generation logic on the server and leveraging templates for each protocol, the API provides a reliable, repeatable, and platform-independent method to produce communication modules tailored to the user's system model.

3.2 Design Rationale

The design and implementation of the code generation tool involved several key decisions regarding its architecture, technology stack, and API protocol. Each choice was guided

[1]The complete open-source implementation of the API is publicly available in a Git repository at: https://github.com/dtavares7/Api

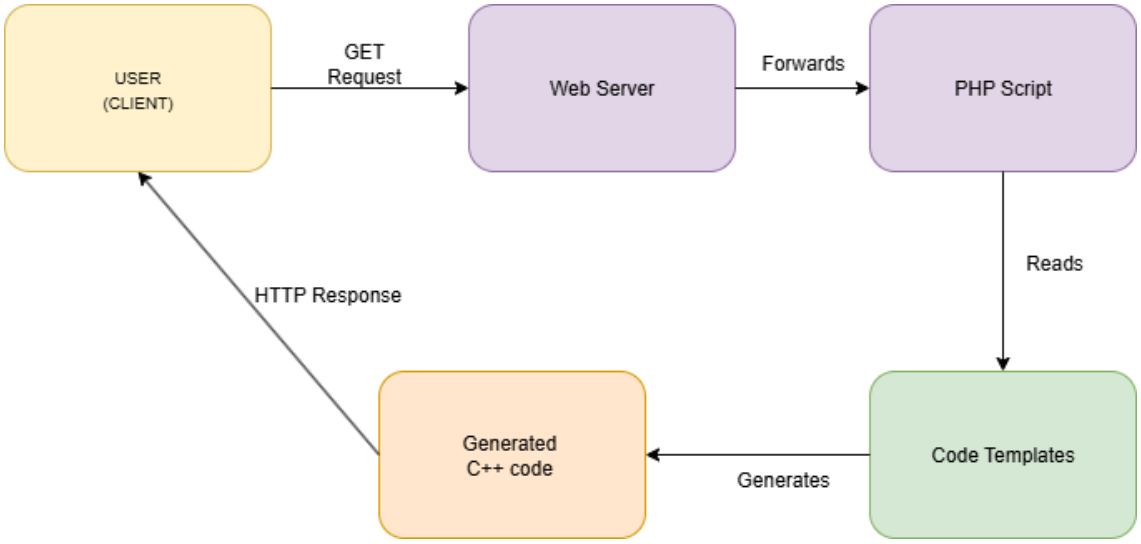

Figure 3.1: The architectural workflow of the code generation API, illustrating the sequential steps from the user request to the generated C++ code response.

by the objectives of platform independence, usability, and seamless integration with the existing IOPT-Tools ecosystem, and is discussed below with a rationale comparing alternative approaches.

The tool was conceived as a server-side Web API rather than a standalone desktop application or a command-line interface. This architectural decision was motivated primarily by the goals of accessibility and platform independence: a Web API can be accessed from any device with a web browser or a standard HTTP client without requiring software installation, whereas desktop applications require separate builds for each operating system, and command-line tools often depend on a specific runtime environment and user familiarity with terminal commands. Furthermore, the Web API architecture facilitates seamless integration with the web-based IOPT-Tools environment, allowing the code generation functionality to be incorporated into the existing front-end workflowdevelopment workflow. Centralized deployment also simplifies maintenance, as updates and bug fixes applied on the server are instantly available to all users, in contrast to desktop or CLI solutions that require distributing updates to individual installations.

The selection of PHP as the implementation language was a deliberate choice after evaluating alternatives such as Python with Flask or FastAPI, and Node.js with Express.While modern frameworks such as Python (with Flask or FastAPI) and Node.js (with Express) were considered for their performance and extensive library ecosystems, the analysis favored PHP for this specific implementation. This decision was primarily driven by the requirement for seamless integration with the existing legacy infrastructure of the IOPT-Tools server, which is natively built on a PHP stack. Adopting a compatible technology stack minimized deployment complexity and ensured interoperability without the need for additional runtime environments or extensive server reconfiguration.

PHP is particularly well-suited for code generation tasks which fundamentally involve sophisticated string manipulation and template processing. Its extensive built-in string handling functions greatly simplify the implementation of the code templating engine, while its server-side simplicity and "shared-nothing" request-response model align naturally with the stateless operation of the API. Additionally, PHP is one of the most widely deployed back-end languages, supported by virtually all web hosting providers, which ensures ease of deployment and long-term maintainability. Another important consideration is the future integration of this API with the IOPT-Tools platform, which itself is implemented in PHP; adopting the same language ensures architectural compatibility and a smooth path for integration, allowing the code generation tool to become a native component of the IOPT-Tools web environment.

Although modern RESTful API design often favors `POST` requests with JSON payloads to handle structured data, a simpler `GET`-based interface was intentionally chosen for this application. This approach offers practical advantages in usability, debugging, and reproducibility: `GET` requests can be easily constructed and tested directly in a web browser's address bar, facilitating rapid experimentation and demonstration, and the tool's dedicated help page (`help.php`) provides additional guidance for users. Since each code generation request is independent, the stateless and idempotent nature of `GET` requests ensures predictable behavior without the need to maintain the server-side session state. Finally, encapsulating the complete API call, including all configuration parameters, within a single URL enhances shareability and allows users to bookmark or distribute specific code generation configurations, as detailed in Section 3.4.

Overall, the combined choice of a server-side Web API implemented in PHP with a `GET`-based interface achieves the project's goals of accessibility, maintainability, and seamless integration. By evaluating alternative approaches and emphasizing usability, the tool provides a robust and user-friendly solution for automated code generation within the IOPT-Tools ecosystem.accessible, offering a lightweight and transparent interface designed to be consumed by client-side applications or automated scripts rather than manual entry.

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 I^2C, `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 I^2C, polling of the serial buffer (`MySerial.available()`) for UART, or a client callback function for MQTT selected over interrupts to prioritize deterministic synchronization within the controller's main execution loop.. 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.Finally, the abstract transfer of tokens between sub-models is directly mapped to the exchange of message payloads, which are configured to carry the unique identifiers (IDs) or signal codes corresponding to the firing transitions. 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. To complement this formal specification and provide a practical, on-demand reference, a dynamic help page was implemented at the `help.php` endpoint. This page serves as a live user manual that presents a summary of the base endpoint, the required global parameters, and a detailed breakdown of each protocol. For each protocol, it lists all available parameters, indicates whether they are required or optional, specifies their default values, and provides a complete, ready-to-use example URL. This feature is designed to facilitate the ease of use and rapid testing of the API directly from a web browser.

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

`http://<server_address>/api.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 MQTT.
`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.

Since I^2C communication operates in a master-slave paradigm, the parameters specified in Table 3.2 are required to ensure proper device addressing and bus management.

Table 3.2: API Parameters for the I^2C 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 acts as the unique identifier for the event. It can be defined as an ASCII character or an integer (0–127), allowing the receiver to distinguish the specific signal source based on this value.

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

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.

3.5 Analysis of Generated Code

This section presents and analyzes the concrete C++ source code produced by the API. The generated code is specifically designed for integration into the `net_io.c` file, a standard output of the IOPT-Tools C code generator that serves as the hardware interface layer.

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_receiver`	Optional	Integer	The RX pin for the device receiving the message. **Default: 16**.
`txPin_receiver`	Optional	Integer	The TX pin for the device receiving the message. **Default: 17**.
`rxPin_sender`	Optional	Integer	The RX pin for the device sending the message. **Default: 17**.
`txPin_sender`	Optional	Integer	The TX pin for the device sending the message. **Default: 16**.
`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>"**.

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>"**.

As the API generates C++ to leverage modern libraries (e.g., for I^2C and MQTT), it is a mandatory step to rename this file to `net_io.cpp` to ensure the C++ compiler is used.

A critical challenge in integrating external communication is bridging the gap between asynchronous events (e.g., the arrival of a network packet or a bus message) and the synchronous, deterministic execution cycle of the Petri net model. To address this, a consistent software design pattern the "asynchronous-to-synchronous bridge"is employed across all generated modules. This pattern uses a `volatile` boolean flag to safely signal the occurrence of an external event to the main synchronous loop. The following subsections

analyze the specific implementation of this pattern for each supported protocol.

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 distinct code for master and slave controllers.
Sample API Call (Slave):

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

The slave controller's code,provided in full in Appendix A, relies on an Interrupt Service Routine (Interrupt Service Routine (ISR)) for event detection. This is an efficient, event-driven approach where the microcontroller's hardware immediately executes the `receiveI²CEvent` function upon data arrival, without requiring the main loop to continuously check for messages.

```
/* PART 1: Global definitions and includes */
#include <Wire.h>
#define I2C_SLAVE_ADDRESS_for_buttonPress 9 // Address for this device
#define CMD_TRIGGER_buttonPress 'P' // Command to activate the event
volatile bool buttonPress_trigger_flag = 0;
void receiveI2CEvent(int byteCount); // Function prototype

/* PART 2: Initialization (inside DemoProject_InitializeIO) */
// Start the i2c bus as a slave with the defined address
Wire.begin(I2C_SLAVE_ADDRESS_for_buttonPress);
// Register the callback function for receiving data
Wire.onReceive(receiveI2CEvent);

/* PART 3: Input Mapping (inside DemoProject_GetInputSignals) */
if (buttonPress_trigger_flag == 1) {
    events->buttonPress = 1;
    buttonPress_trigger_flag = 0; // Reset flag to fire only once
} else {
    events->buttonPress = 0; // Ensure event is inactive by default
}

/* PART 4: Interrupt Service Routine (Helper Functions) */
void receiveI2CEvent(int byteCount) {
    if (Wire.available() > 0) {
        char command = Wire.read();
        if (command == CMD_TRIGGER_buttonPress) {
            buttonPress_trigger_flag = 1;
        }
    }
    // Flush remaining buffer
    while (Wire.available() > 0) {
        Wire.read();
```

```
33     }
34 }
```

Listing 3.1: Generated C++ code structure for I2C Slave event reception

The core of the asynchronous-to-synchronous bridge is the volatile bool buttonPress_trigger_flag. The `volatile` keyword is critical: it instructs the compiler not to apply optimizations to this variable, ensuring that every read of the flag in the main loop fetches its true, up-to-date value from memory, which may have been modified at any time by the ISR. This prevents potential race conditions and ensures the event is reliably detected by the synchronous Petri net logic.As illustrated in Listing 3.1, the generated code is organized into four distinct functional blocks to ensure proper integration with the IOPT execution cycle.

Lines 1–6 define the global scope, including the necessary library headers, the device's unique I^2C address (line 3), and the specific command character acting as the event identifier (line 4). The core of this asynchronous-to-synchronous bridge is the `volatile` boolean flag declared in line 5. The `volatile` keyword is critical: it instructs the compiler not to apply optimizations to this variable, ensuring that every read of the flag in the main loop fetches its true, up-to-date value from memory, which may have been modified at any time by the ISR. This mechanism prevents potential race conditions and ensures the event is reliably detected by the synchronous Petri net logic.

The initialization logic is depicted in lines 8–12, where the device joins the I^2C bus as a slave and registers the `receiveI2CEvent` function as the callback for incoming data interrupts.

The synchronization between the asynchronous reception and the synchronous model execution occurs in lines 14–20. Here, inside the input reading function, the code checks the volatile flag; if set, it activates the corresponding IOPT event (line 16) and immediately resets the flag (line 17) to prevent duplicate event generation in subsequent cycles.

Finally, lines 23–34 contain the interrupt service routine. This function is triggered automatically by the hardware when data is received. It reads the incoming command byte (line 25), validates it against the expected trigger character (line 26), and sets the flag (line 27) without blocking the main processor, ensuring efficient event handling.

3.5.2 UART-Based Communication Channel

UART communication is ideal for point-to-point serial data exchange. Unlike the ISR-driven I^{2}C slave, the UART implementation uses polling.

Sample API Call:

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

In the code, provided in full in Appendix B, the main loop actively checks for incoming data in each cycle by calling `MySerial.available()`. This polling approach is simpler to

implement than ISRs and avoids some complexities of interrupt programming. However, it relies on the Petri net's execution cycle being fast enough to check the serial buffer before it overflows. This presents a classic trade-off in embedded systems: ISRs offer lower latency and higher responsiveness at the cost of complexity, while polling is simpler but can potentially miss events in systems with a slow or variable loop rate.

```
/* PART 1: Global definitions and includes */
#include <HardwareSerial.h>

// -- UART Configuration --
HardwareSerial MySerial(2);
#define RXD2 16
#define TXD2 17

// -- Message and Flag Definitions --
const String message_uart_toggleLED = "trigger_toggleLED";
volatile bool toggleLED_trigger_flag = 0; // flag only for receiving signals

// -- Function Prototypes --
void setupUart_toggleLED();
String receiveDataUart();
void waitMessageUart();

/* PART 2: Initialization (inside SystemA_InitializeIO) */
setupUart_toggleLED();

/* PART 3: Input Mapping (inside SystemA_GetInputSignals) */
waitMessageUart(); // Explicit polling call
if (toggleLED_trigger_flag == 1) {
    events->toggleLED = 1;
    toggleLED_trigger_flag = 0; // Reset the flag to fire only once
} else {
    events->toggleLED = 0; // Ensure the event is inactive by default
}

/* PART 4: Helper Functions */
void setupUart_toggleLED() {
    MySerial.begin(9600, SERIAL_8N1, RXD2, TXD2);
    Serial.println("UART communication initialized (Receiver).");
}

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

```

```
45 void waitMessageUart() {
46     String receivedMessage = receiveDataUart();
47     if (receivedMessage.length() > 0 && receivedMessage == message_uart_toggleLED) {
48         toggleLED_trigger_flag = 1;
49         Serial.println("Correct␣message␣received.␣Flag␣activated.");
50     }
51 }
```

Listing 3.2: Generated C++ code structure for UART Event Reception

Despite the different detection mechanism (polling vs. ISR), the code employs the exact same `volatile` flag pattern to safely bridge the asynchronous nature of serial data arrival with the synchronous `GetInputSignals()` function call.

Listing 3.2 demonstrates the generated code for UART communication. While structurally similar to the I^2C example regarding variable scope and flag usage, the execution flow differs significantly due to the polling-based nature of the serial implementation.

Lines 1–14 establish the global environment. A dedicated `HardwareSerial` instance is created (line 5) with specific RX/TX pin definitions (lines 6–7), enabling communication without interfering with the main USB serial port. The target message string is defined in line 10. Crucially, line 11 declares the `volatile` boolean flag, maintaining the same thread-safety pattern used in other protocols to prevent compiler optimizations from caching the variable value.

The initialization in Part 2 (line 17) calls the setup function defined in lines 28–31, which configures the baud rate and data frame format (8N1).

The most distinct mechanism appears in Part 3 (lines 20–26). Unlike the interrupt-driven I^2C example, this code explicitly calls `waitMessageUart()` (line 20) at the beginning of the input cycle. This function, detailed in lines 42–48, polls the serial buffer, reads any available strings, compares them against the expected trigger message, and updates the flag if a match is found.

Finally, lines 21–26 execute the bridge logic: checking the flag state and mapping it to the IOPT event variable. Despite the different detection mechanism (polling versus ISR), this confirms that the code employs the exact same `volatile` flag pattern to safely bridge the asynchronous nature of serial data arrival with the synchronous `GetInputSignals()` function call.

3.5.3 TCP/MQTT-Based Communication Channel

For networked systems, the tool generates code using the MQTT protocol, which enables a robust publish-subscribe model. This approach is inherently asynchronous and event-driven.

Sample API Call:

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

The implementation, provided in full in Appendix C, leverages the `PubSubClient` library, which uses a callback function (`callback`) that is automatically executed when a message arrives on a subscribed topic. This is conceptually similar to an ISR but operates at a higher software level. The same asynchronous-to-synchronous bridge pattern is used to signal the event to the main loop.

```
/* PART 1: Global definitions and includes */
#include <WiFi.h>
#include <PubSubClient.h>

// Global Objects (Generated based on config)
WiFiClient espClient;
PubSubClient client(espClient);
volatile bool alert_trigger_flag = 0; // The bridge pattern again

/* PART 3: Input Mapping (inside SensorNetwork_GetInputSignals) */
if (alert_trigger_flag) {
    events->alert = 1;
    alert_trigger_flag = 0; // Reset flag to ensure single firing
} else {
    events->alert = 0;
}

/* PART 4: Network Maintenance (inside SensorNetwork_LoopDelay) */
// Crucial for maintaining connection and processing incoming packets
loopDelayTcp(topic_sub_alert);

/* PART 5: Helper Functions */
void tcpMqttInitializeIO() {
    Serial.begin(115200);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    client.setServer(mqtt_broker, mqtt_port);
    client.setCallback(callback);
}

void reconnect(const char* topic) {
    while (!client.connected()) {
        Serial.print("Attempting␣MQTT␣connection...");
        if (client.connect(client_id)) {
            Serial.println("connected");
            client.subscribe(topic);
        } else {
            Serial.print("failed,␣rc=");
            Serial.println(client.state());
            delay(5000);
        }
```

```
45     }
46 }
47
48 void loopDelayTcp(const char* topic) {
49     if (!client.connected()) {
50         reconnect(topic);
51     }
52     client.loop(); // Processes incoming messages and calls callback
53 }
54
55 void callback(char* topic, byte* payload, unsigned int length) {
56     String msg = "";
57     for (int i=0; i<length; i++) msg += (char)payload[i];
58     msg.trim();
59     if (msg == String(message_tcp_alert)) {
60         alert_trigger_flag = 1;
61         Serial.println("Correct TCP message received. Flag activated.");
62     }
63 }
```

Listing 3.3: Generated C++ code structure for MQTT Event Reception

A key feature of the generated MQTT code is its focus on resilience. Distributed systems must be able to handle network disruptions. The generated code includes a `reconnect()` function that automatically attempts to re-establish the connection to the Wi-Fi network and the MQTT broker if it is lost. This ensures that the controller can recover from transient network failures, making the overall system more robust.

Listing 3.3 presents the generated code for the MQTT implementation. Due to the complexity of maintaining a network stack, this module requires a more robust structure than the serial protocols.

Lines 1–7 set up the environment, including the necessary Wi-Fi and MQTT client libraries. Consistent with the previous examples, a `volatile` boolean flag (line 7) is employed to bridge the asynchronous network callbacks with the synchronous Petri net execution.

The reception logic is distributed across two key areas. First, the `callback` function (lines 53–61) is automatically invoked by the library whenever a message arrives on a subscribed topic. It reconstructs the payload string and, if it matches the trigger message, raises the flag. Second, the synchronous reading of this flag occurs in lines 10–15, mapping the network event to the IOPT signal.

A critical addition in this template is Part 4 (lines 18–19), which calls `loopDelayTcp()`. Unlike hardware interrupts, the MQTT client requires regular processing cycles to handle network traffic and keep-alive signals. This function (defined in lines 46–51) ensures that `client.loop()` is called during the controller's idle time.

Finally, a key feature of the generated MQTT code is its focus on resilience, as highlighted in the `reconnect()` function (lines 33–44). Distributed systems must be able to

handle network disruptions. This function checks the connection state and, if lost, enters a blocking loop to automatically attempt to re-establish the connection to the MQTT broker and re-subscribe to the relevant topics. This ensures that the controller can recover from transient network failures, making the overall system robust and suitable for real-world deployment.

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 and robustness against a range of inputs, 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 3.4. The methodology involved subjecting the API to a series of HTTP GET requests with both valid (positive testing) and invalid (negative testing) parameter sets to verify correct behavior and error handling. A representative sample of these test cases is detailed in Table 3.5.

The tests confirmed that the tool consistently produced syntactically correct C++ code for all valid parameter combinations. In all negative test cases, the API correctly identified the input as invalid and returned the expected HTTP 400 status code, confirming 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 in a real-world scenario.

3.6.2.1 Testbed Environment

A physical testbed consisting of two ESP32-WROOM-32 microcontroller development boards was established. The development environment was the Arduino IDE, utilizing standard libraries such as `Wire.h` for I^2C, `HardwareSerial.h` for UART, and `PubSubClient.h` for MQTT. For network tests, the boards connected via Wi-Fi to a local network, which had access to a public MQTT broker (HiveMQ). Communication was monitored via the Arduino IDE's serial monitor.

Table 3.5: Representative API Test Cases and Results

Test Type	Description	Sample Input (URL Fragment)	Expected Outcome
Positive	Valid I²C request with all mandatory parameters.	`?protocol=I2C&projectName=Demo &eventName=e1&slaveAddress=8 &slaveMessage=A`	HTTP 200 OK with syntactically correct C++ code.
Positive	Valid UART request using default and custom parameters.	`?protocol=uart&projectName=Demo &eventName=e2&baudRate=9600`	HTTP 200 OK with syntactically correct C++ code.
Positive	Valid TCP/MQTT request with a specific topic.	`?protocol=tcp&projectName=Demo &eventName=e3&topic=dev/test`	HTTP 200 OK with syntactically correct C++ code.
Negative	Missing a mandatory parameter (`topic` for TCP).	`?protocol=tcp&projectName=Demo &eventName=e3`	HTTP 400 Bad Request with a descriptive error message.
Negative	Parameter value outside the valid range (`slaveAddress` > 119).	`?protocol=I2C&projectName=Demo &eventName=e1&slaveAddress=150`	HTTP 400 Bad Request with a descriptive error message.
Negative	Unsupported protocol name.	`?protocol=can&projectName=Demo &eventName=e4`	HTTP 400 Bad Request with a descriptive error message.

3.6.2.2 Validation Procedure and Results

For each of the three protocols, a pair of sender (Master/Publisher) and receiver (Slave/-Subscriber) applications was generated using the API. The code was integrated into a minimal Arduino project, compiled, and uploaded to the two ESP32 boards. The boards were connected according to the protocol requirements. The sending device was then triggered to simulate a Petri net output event, and the receiving device's serial monitor was observed for confirmation of message reception. The results of these empirical tests are summarized in Table 3.6.

The empirical tests were successful for all three communication protocols. A crucial aspect of this validation was to confirm the seamless integration between the generated communication module and the synchronous Petri net execution core. In every test scenario, after a message was received, the event was correctly registered as an input signal by the Petri net's `GetInputSignals()` function. This new input subsequently enabled the firing of the appropriate transitions within the model, causing the controller to advance its state as specified by the IOPT logic. This successfully demonstrates that the generated

Table 3.6: Summary of Generated Code Validation Tests

Protocol	Test Scenario (Sender Action)	Expected Receiver Behavior	Result
I²C	The master controller sends a specific command byte to the slave's address.	The slave's I²C `onReceive` ISR is triggered, correctly identifies the command, and sets the corresponding `volatile` flag. A confirmation message is printed to the serial monitor.	Pass
UART	The sending controller transmits a predefined string message over the serial TX line.	The receiving controller, polling its RX line, reads the complete string, validates its content, and sets the corresponding `volatile` flag. A confirmation message is printed.	Pass
TCP/MQTT	The publisher client sends a specific message payload to a predefined topic on the MQTT broker.	The subscriber client, connected to the same broker and subscribed to the topic, receives the message via its callback function and sets the corresponding `volatile` flag. A confirmation message is printed.	Pass

modules provide not only a reliable communication channel but also a functionally correct interface that bridges the asynchronous external world with the deterministic execution of the Petri net.

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.

To achieve this, a case study is presented centered on a distributed controller for a three-conveyor automation system. 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 [15]:

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.

2. **To conduct a quantitative performance analysis:** By implementing intercontroller communication using three distinct protocols, I^2C, 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.evaluating both Flash memory (program storage) and SRAM usage (dynamic data allocation). This breakdown offers valuable insight into the trade-offs associated with each protocol in a resource-constrained embedded environment, where efficient memory management is critical to ensure that communication stacks do not compromise the stability of the core IOPT control logic.

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.

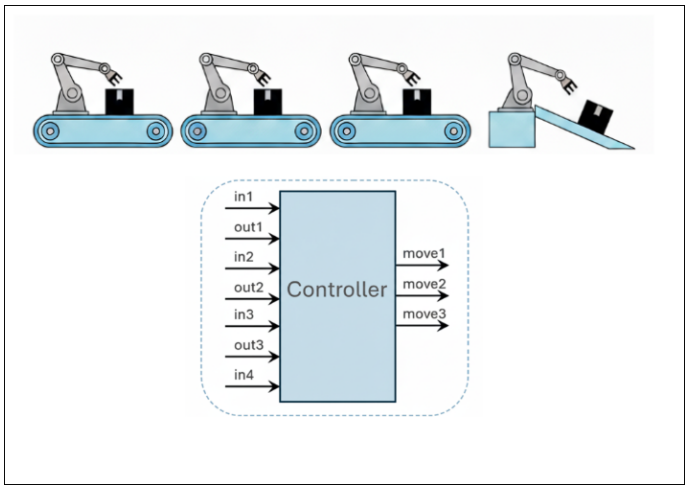

Figure 4.1: Layout of the three-conveyor system, showing the placement of sensors and the direction of item flow, adapted from [15].

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,

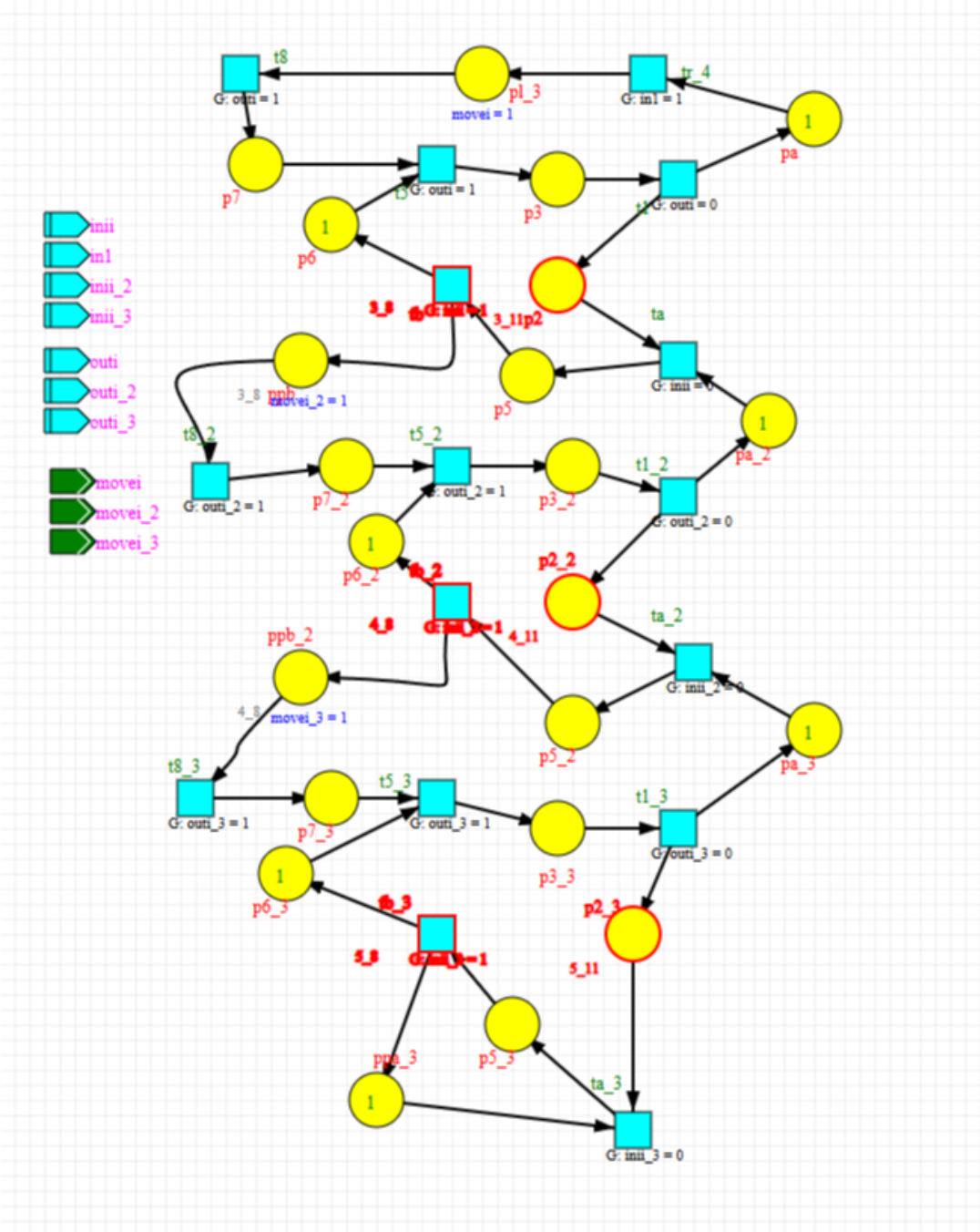

Figure 4.2: The global IOPT Petri net model specifying the centralized controller for the entire conveyor system. The nodes highlighted in red represent the chosen cutting set for decomposition, adapted from [15].

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.

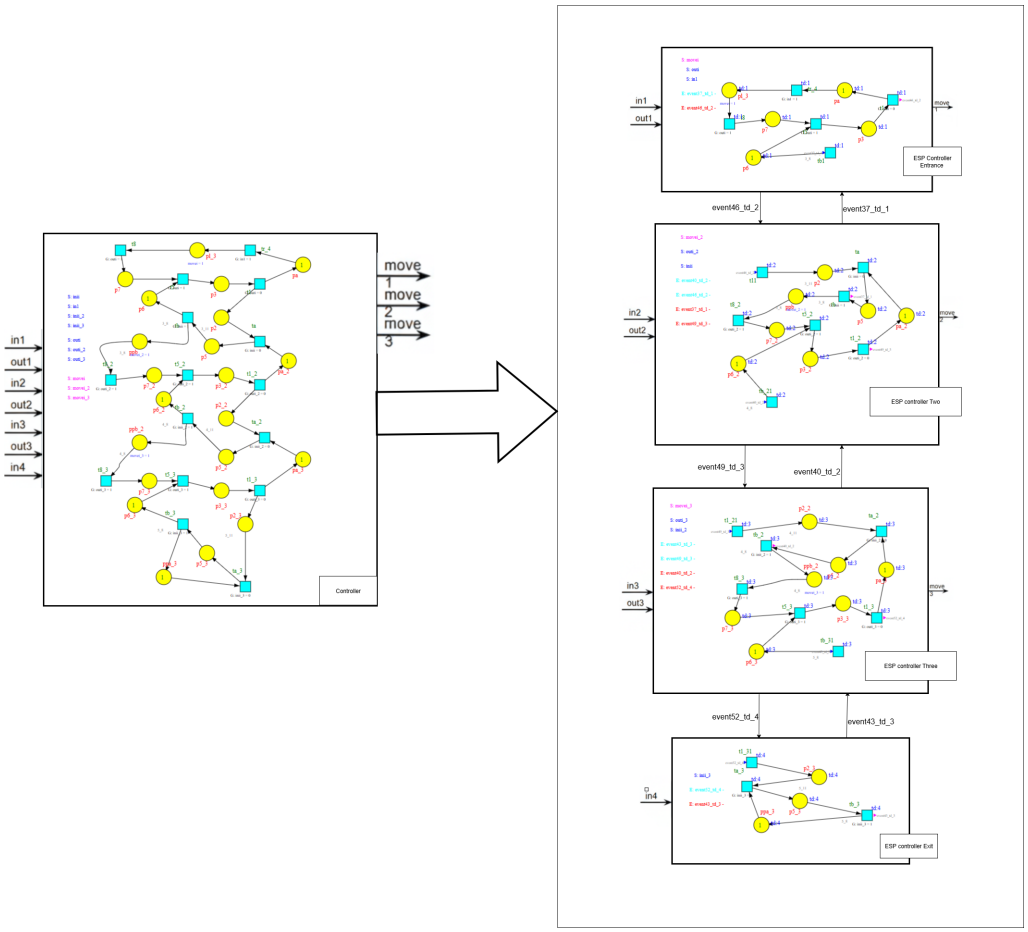

Figure 4.3: The transformation from the centralized controller model (left) to a set of concurrent, networked sub-models (right) resulting from the net splitting operation, adapted from [15].

4.3 Implementation and Functional Validation

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.

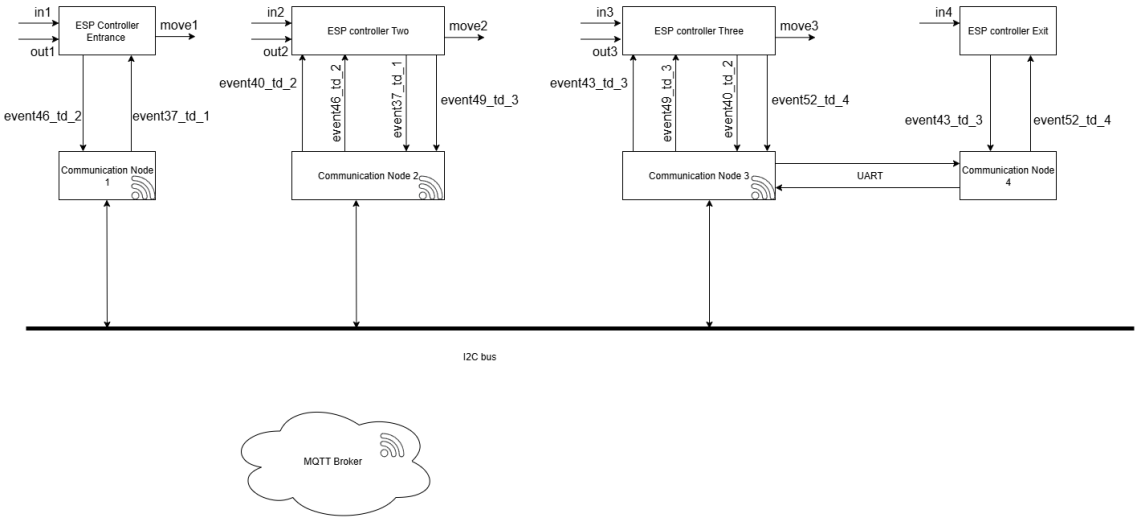

Figure 4.4: The heterogeneous deployment architecture for the distributed conveyor system controller, utilizing I^{2}C, UART, and MQTT over TCP/IP for inter-controller communication and synchronization, adapted from [15].

4.3.1 I^{2}C Bus Implementation

As depicted in the deployment architecture (Figure 4.4), *Controller Two* acts as the master on a local I^{2}C 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, two API calls similar to the following were used:

```
?protocol=i2c&projectName=ControllerTwo&eventName=event37_td_1
&slaveAddress=8&slaveMessage=A
```

And for the second event:

```
?protocol=i2c&projectName=ControllerTwo&eventName=event49_td_3
&slaveAddress=9&slaveMessage=B
```

To validate the functional behavior of the generated I^{2}C module, the execution of *Controller Two* was captured using the IOPT-Tools simulator. The resulting timing diagram, presented in Figure 4.5, provides a detailed view of the controller's operational logic and illustrates the direct correlation between the external I^{2}C communication events and the internal state of the Petri net.

The behavior can be analyzed by observing the two primary I^{2}C messages:

- **Event `event37_td_1`:** The sequence shows an internal state change, where the marking of a place (e.g., `p_11/ppb`) enables the firing of a transition (e.g., `t_27/t11`). The diagram confirms that the firing of this transition directly causes the generation of the output signal `event37_td_1`. This corresponds to *Controller Two*, acting as the I^{2}C master, sending a command to one of its slaves (such as *Controller Entrance*).

- **Event `event49_td_3`:** Later in the execution cycle, after the physical process has evolved (indicated by changes in signals like `outi_2`), the Petri net reaches a new state. This leads to another transition firing, which in turn causes the generation of the second I²C message, `event49_td_3`. This represents *Controller Two* sending a subsequent command to another slave (such as *Controller Three*).

This analysis confirms that the API-generated I²C communication module functions correctly and that the deployed hardware implementation faithfully reproduces the semantics of the original IOPT model.

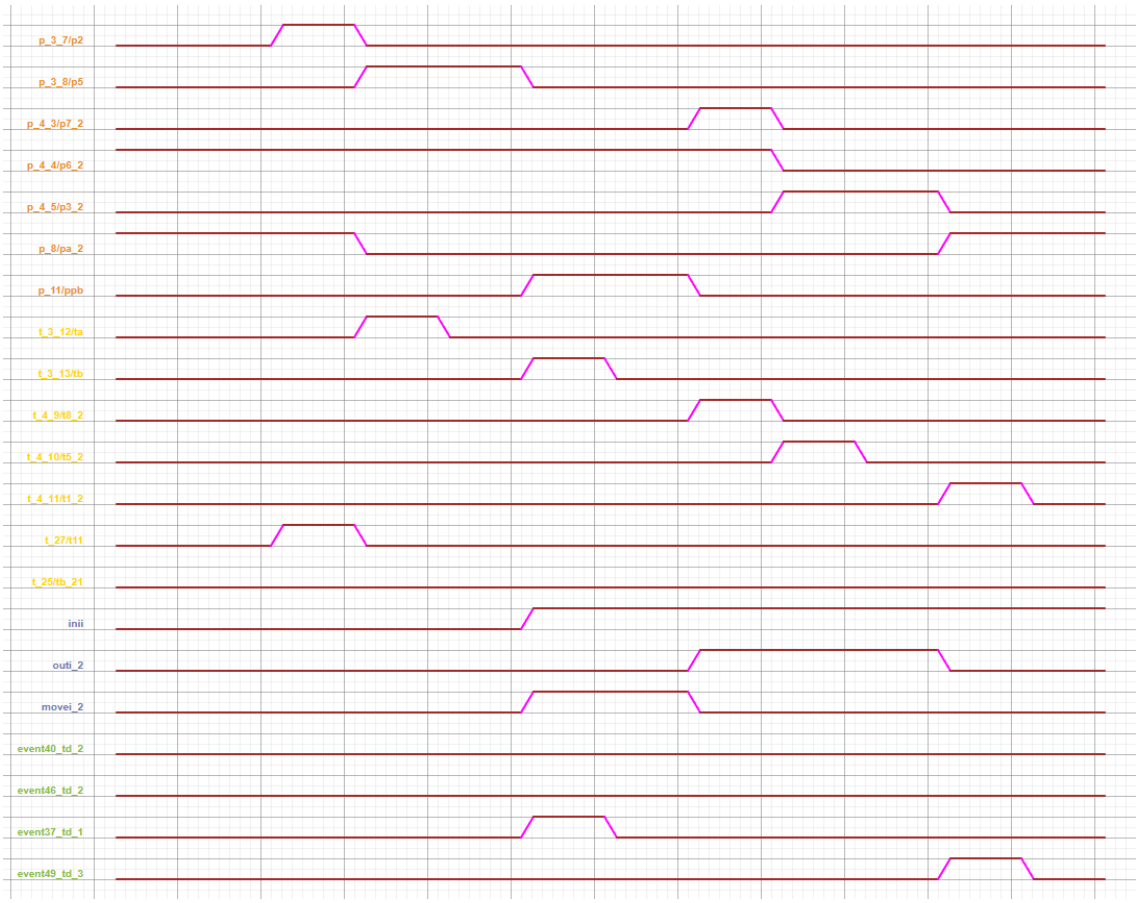

Figure 4.5: Timing diagram for *Controller Two*, showing how internal Petri net states (p_...) and transition firings (t_...) lead to the generation of I²C output events (`event37_td_1`, `event49_td_3`).

This complete validation of the I²C module demonstrates the tool's ability to correctly implement the master-slave bus communication as specified by the IOPT model.

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=ControllerExit
&eventName=event43_td_3&baudRate=115200
```

To validate the sending side of the UART link, the behavior of *Controller Three* was analyzed using a timing diagram captured by the IOPT-Tools simulator. The diagram, shown in Figure 4.6, details the internal Petri net logic that leads to the transmission of the UART message.

The analysis of the diagram is as follows.

- The operational sequence begins with the firing of an internal transition, **`t_29/t1_31`**, which corresponds to the arrival of the event `event53_td_4`, and then advances the Petri net to a state where it awaits a physical input.
- The controller remains in this state until the physical sensor input **`inii_3`** is asserted. This external event enables the final transitions in the sequence (`t_5_12/ta_3` and `t_5_13/tb_3`).
- Critically, the firing of these final transitions culminates in the generation of the output signal **`event43_td_3`**. This represents the moment *Controller Three* transmits the message payload over the UART link to *Controller Exit*.

This analysis confirms that the API-generated UART sender code functions as specified. It demonstrates that the controller's internal logic, driven by both its Petri net state and physical inputs, correctly results in the transmission of the serial message at the appropriate time in the operational cycle.

4.3.3 TCP/MQTT-Based Communication Channel

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

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

To validate network-level publisher functionality, the behavior of one of the MQTT-enabled controllers (Controller Three) was analyzed using the IOPT-Tools simulator. The timing diagram, shown in Figure 4.7, details how the controller's internal logic, initiated by a generic input event, culminates in the transmission of a message to the MQTT broker.

The analysis of the diagram is as follows.

- **Cycle Trigger:** The operational sequence is initiated by the arrival of an external event from another controller. The reception of this event is represented by the firing of the initial transition, **`t_28/t1_21`**. This trigger advances the Petri net's state and prepares the controller for its main task.

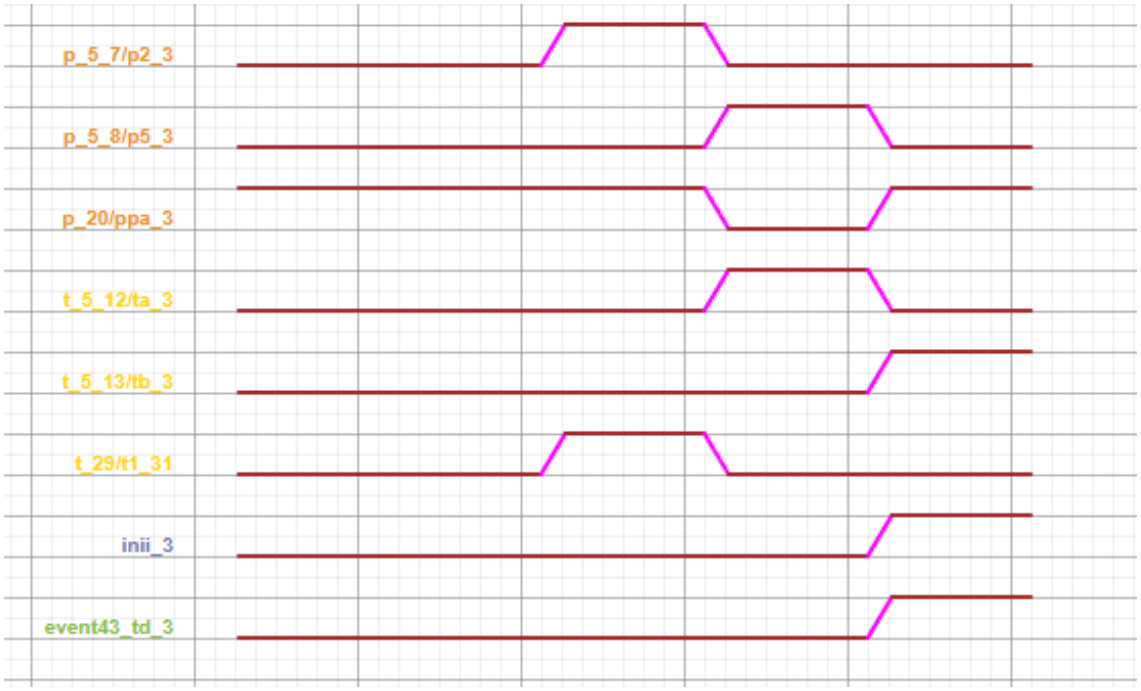

Figure 4.6: Timing diagram for *Controller Exit*, showing internal state changes and a physical input (`inii_3`) lead to the transmission of a UART message (`event43_td_3`).

- **Message Transmission (Publish):** After a sequence of internal state changes that are also dependent on physical inputs (like **`inii_2`**), the Petri net's logic enables the final transition, **`t_4_13/tb_2`**. The diagram confirms that the firing of this transition correctly causes the generation of the output signal **`event40_td_2`**, which represents the controller publishing its message to the specified MQTT topic.

This analysis validates that the API-generated TCP/MQTT module's publisher functionality is correctly integrated with the Petri net logic, sending network messages to the broker at the appropriate time based on the controller's state and inputs.

The successful validation of all three communication modules confirms the tool's ability to reliably implement a complex, heterogeneous communication architecture as specified by the IOPT model.

4.3.4 Code Integration

For each controller, the baseline logic code was first generated using the standard IOPT-Tools generator, creating a `net_io.c` file. **As a mandatory first step, this file was renamed to `net_io.cpp` to enable C++ compilation, a requirement for the libraries used by the generated communication modules.** Subsequently, the specific communication modules produced by the API calls were inserted directly into this renamed `net_io.cpp` file, completing the hardware-specific implementation. The entire process of specifying, generating, and integrating the code for all three heterogeneous communication links was

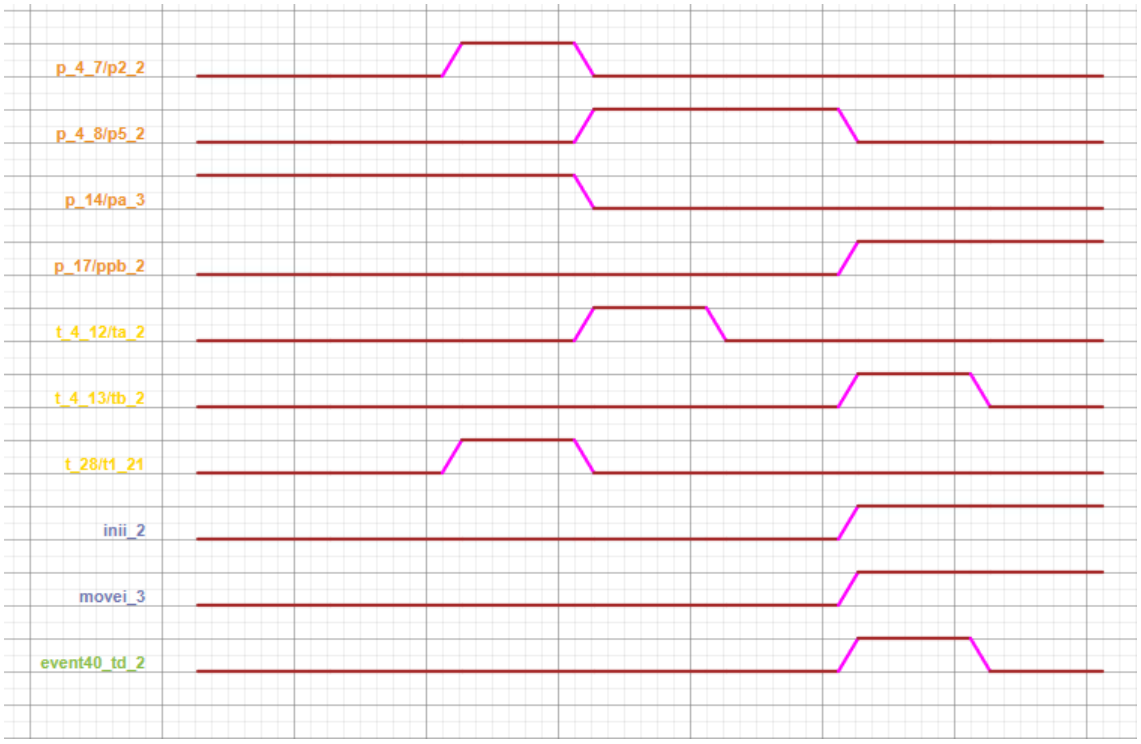

Figure 4.7: Timing diagram validating the MQTT publisher. An external input event triggers a Petri net sequence that results in the transmission of an MQTT message (`event40_td_2`).

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 Experimental Setup and Methodology

After successfully implementing the distributed controller for the conveyor system, a comprehensive experimental analysis was performed. This evaluation aims to both quantify the performance overhead and validate the functional correctness of the tool's generated code. Specifically, this analysis focuses on three critical aspects: the **memory footprint**, as a key constraint in embedded hardware; the **communication latency**, which directly impacts system responsiveness; and the **dynamic behavior**, to ensure the deployed code correctly implements the formal model's semantics.

4.4.1 Hardware Testbed

To empirically validate the generated code and conduct the performance analysis, a physical hardware testbed was constructed, representing the distributed controller architecture shown in Figure 4.4.

The testbed, photographed in Figure 4.8, consists of four ESP32-WROOM-32 development boards. These boards were interconnected using a combination of direct wiring and a shared bus on a breadboard to physically realize the I^2C and UART communication links. The TCP/MQTT communication was facilitated by the built-in Wi-Fi capabilities of the ESP32s, which allows them to connect to a shared network. This physical prototype served as the basis for all the validation and performance measurements presented in this chapter.

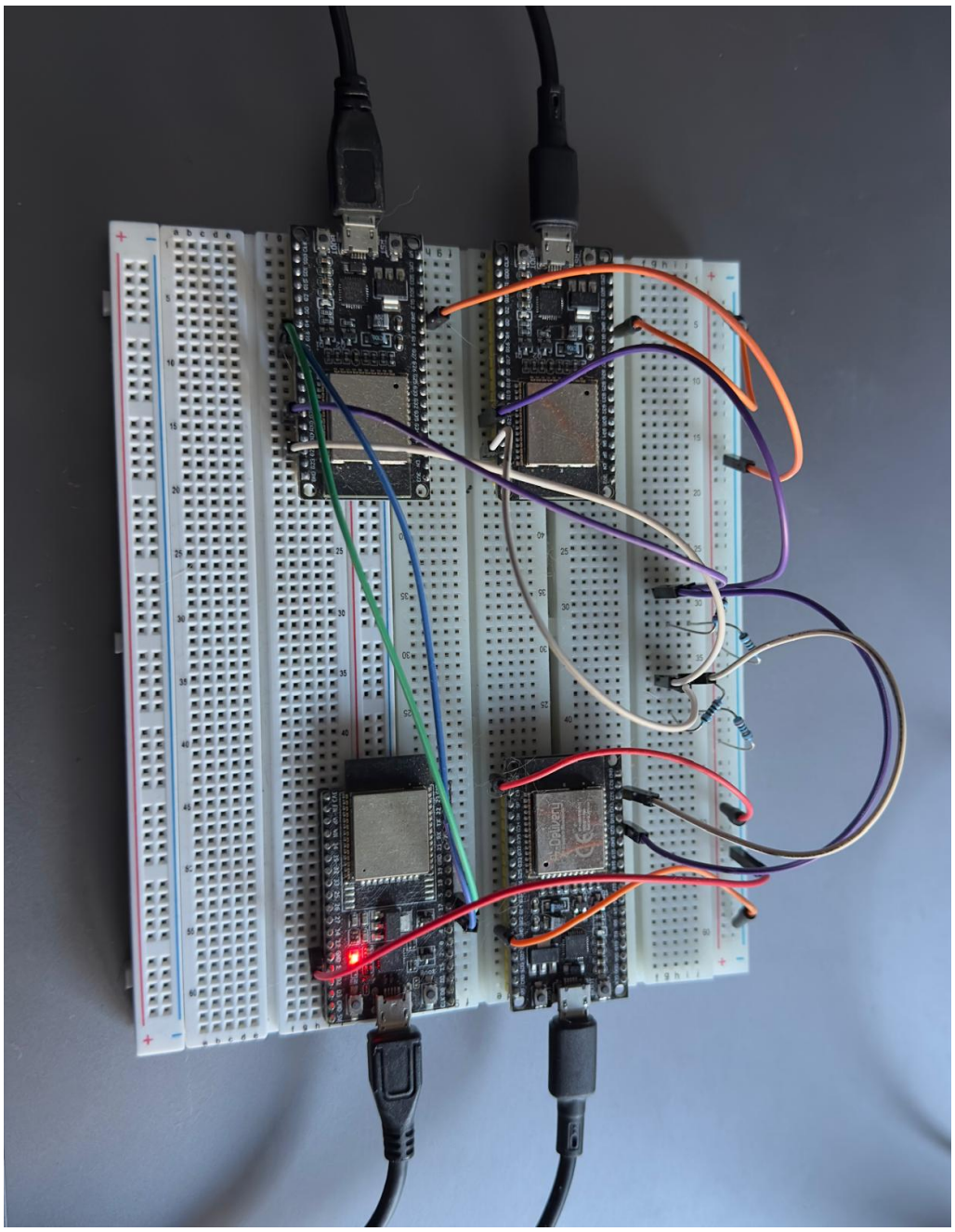

Figure 4.8: The physical hardware testbed for the three-conveyor system, showing the four interconnected ESP32 controller modules.

4.4.2 Analysis Methodology

To evaluate the system's performance and functional correctness, three distinct analyses were conducted. The methodology for each is detailed below.

4.4.2.1 Memory Footprint Measurement

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.

4.4.2.2 Communication Latency Measurement

The latency for each protocol was measured empirically on the hardware testbed. The primary tool for this measurement was the ESP32's high-resolution `micros()` function, which provides microsecond-level timing accuracy. The output values were observed using the Arduino IDE's **Serial Monitor**, connected to the receiving controller via USB.

The specific procedure varied for each protocol to best capture its characteristics:

- **I^2C and UART:** To measure the direct one-way transfer time, the sending controller recorded a start timestamp, `t_start = micros()`, and embedded this value within the message payload itself. Upon reception, the receiving controller immediately recorded an end timestamp, `t_end = micros()`. The latency was then calculated as `latency = t_end - t_start` and printed to the Serial Monitor.

- **TCP/MQTT:** Due to the indeterminate path through a public broker, a round-trip time (RTT) was measured to provide a stable metric. A "ping-pong" test was configured where the first controller published a message and recorded `t_start`. The second controller was programmed to immediately publish a reply upon receiving the message. The first controller recorded `t_end` upon receiving the reply. The one-way latency was then estimated as `RTT/2`.

4.5 Results and Discussion

The methodologies described in the previous section were applied to the hardware testbed, yielding quantitative results regarding performance and functional correctness. These results are presented and analyzed below.

4.5.1 Performance Analysis

4.5.1.1 Memory Footprint Overhead

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, adapted from [15].

Controller ESP	Baseline (Bytes)	W/ Comm. (Bytes)	Overhead (Bytes)	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

4.5.1.2 Communication Latency

The application of the latency measurement methodology yielded the average latency values presented in Table 4.2. These results, visualized in Figure 4.9, highlight the orders-of-magnitude differences in temporal performance between the protocols.

Table 4.2: Average Communication Latency Measurements.

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

For each protocol, the test was executed **20 times** to account for minor variations, and the average of these measurements was taken as the final, representative result.

4.5.2 Discussion and Trade-off Analysis

The performance data, encompassing both memory footprint (Table 4.1) and latency (Table 4.2), provides a clear and quantitative basis for analyzing the engineering trade-offs associated with each communication protocol.

The analysis confirms that hardware-level protocols are exceptionally efficient. The **Exit Controller**, using only UART, shows a negligible memory overhead of just 1.4 kB

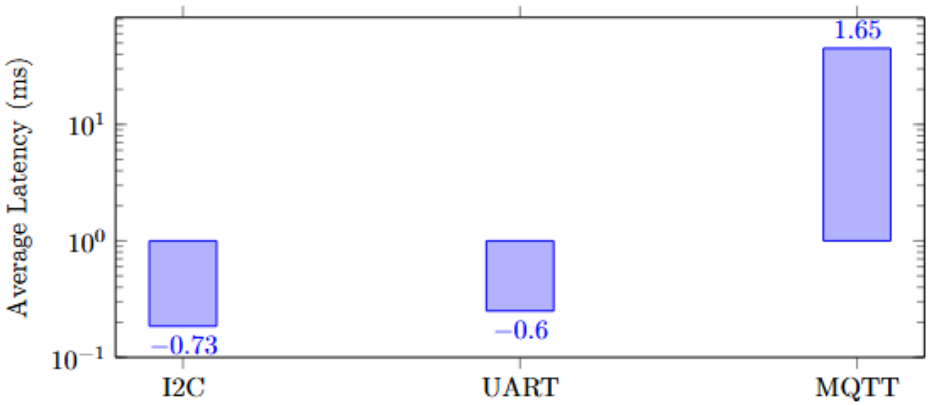

Figure 4.9: Comparison of average communication latency across protocols. Note the logarithmic scale on the y-axis is required to visualize the vast difference in performance.

and a measured latency of 250 µs. This extremely low latency is a direct result of the hardware-based, point-to-point nature of the protocol. A similar high performance is seen with I^2C, which benefits from a synchronous, hardware-managed bus.

Conversely, the controllers implementing MQTT exhibit a substantial memory overhead of over 26 kB and a significantly higher latency measured at 45 ms, over 180 times slower than UART. This result is expected and is attributable to the multiple layers of abstraction and physical infrastructure involved: the TCP/IP and Wi-Fi stack, wireless transmission, and the round-trip delay to the public MQTT broker.

These results underscore a fundamental design trade-off: **resource efficiency versus functional flexibility**. While I^2C and UART are highly efficient, they are limited to local, physically-wired communication. MQTT, despite its resource-intensiveness, offers unparalleled flexibility, enabling communication over vast distances, easy integration with cloud services, and a scalable publish-subscribe architecture.

The model-driven approach, augmented by the tool presented in this thesis, empowers a designer to manage these trade-offs intelligently. By partitioning a system and analyzing the specific requirements of each inter-controller link, a designer can make informed, data-driven decisions. High-speed, critical control loops can be implemented with low-latency protocols like I^2C, while non-critical status updates or remote monitoring can leverage the flexibility of MQTT, all within the same distributed system. This allows for an optimized heterogeneous architecture that meets both functional requirements and hardware constraints.

4.6 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 here 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 I^2C, UART, and TCP/MQTT protocols.

Furthermore, the performance analysis provided critical quantitative insight into the resource consumption and temporal characteristics of these protocols. The results empirically confirmed the significant trade-offs in **memory overhead and communication latency** between network-level protocols like TCP/MQTT and the lightweight, high-speed nature of hardware-level protocols such as UART and I^2C . 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

5.1 Summary of the Work

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 I^2C, 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 and communication latency** associated with each protocol, offering empirical data on the trade-offs between them.

5.2 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 (I^2C, 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.

Another 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.

In addition, this work presents a novel empirical analysis of the communication modules generated, providing quantitative performance data to support protocol selection. By measuring key performance metrics, including the **memory footprint and communication latency** of each protocol within a real-world application, this study 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.

Additionally, the public release of the source code of the developed software tool promotes transparency and enables the reproducibility of the results presented in this dissertation. The implementation is available in a public Git repository[1].

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 completecomprehensive and validated workflow, supported by empirical evidence, that can guide both researchers and practitioners in the design of reliable and efficient distributed control solutions.

5.3 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: I^2C, UART, and TCP/MQTT. Although these protocols cover a range of typical use cases, from

[1]The complete open-source implementation of the API is available at: https://github.com/dtavares7/Api

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.

5.4 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.

5.5 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. 15).

[2] A. Banks and R. Gupta. "MQTT: Message Queuing Telemetry Transport". In: *Proceedings of the 3rd International Conference on Internet of Things*. IEEE, 2014, pp. 590–596. DOI: 10.1109/iThings.2014.83 (cit. on p. 17).

[4] 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. 15).

[5] 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. 10, 18, 19).

[6] M. Broy, M. V. Cengarle, and E. Geisberger. "Cyber-Physical Systems: A Tectonic Shift?" In: *Fascination of Software, Systems and Services*. Ed. by M. Broy and E. Denert. Springer Berlin Heidelberg, 2012, pp. 3–19. ISBN: 978-3-642-27373-5. DOI: 10.1007/978-3-642-3fascination-s-3_1 (cit. on p. 2).

[7] 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 pp. 9, 10).

[8] 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 pp. 9, 10).

[9] B. A. Forouzan. *TCP/IP Protocol Suite*. 4th. New York, NY: McGraw-Hill, 2010 (cit. on p. 16).

[10] 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 p. 9).

[11] A. Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys & Tutorials* 17.4 (2015), pp. 2347–2376. DOI: 10.1109/COMST.2015.2444095 (cit. on p. 1).

[12] 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. 5).

[13] 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. 8).

[17] L. Gomes et al. "Merging and Splitting Petri Net Models within Distributed Embedded Controller Design". In: *Embedded Systems and Robotics with Open Source Technologies*. Ed. by I. V. P. T. S. Hristu-Varsakelis. IGI Global, 2012, pp. 153–166. ISBN: 9781466639232. DOI: 10.4018/978-1-4666-3922-5.ch009 (cit. on pp. 9, 18, 19, 24).

[14] 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. 18, 21).

[15] L. Gomes and D. Tavares. "Integrating Heterogeneous Communication Support in Model-Driven Development of Petri nets based Distributed Controllers". In: *Proceedings of the 2026 IEEE/SICE International Symposium on System Integration (SII)*. (Submitted and under review.) IEEE, 2026 (cit. on pp. 40–44, 51).

[16] 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. 20, 22).

[18] 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. 7, 8, 18–20, 24).

[19] 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. 18, 24).

[22] GRES/UNINOVA. "IOPT Tools User Manual". In: GRES - Research Group on Embedded Systems (UNINOVA). URL: http://gres.uninova.pt/ (visited on 2025-03-14) (cit. on pp. 20, 22, 23).

[20] Hannes Siebeneicher. *Universal Asynchronous Receiver-Transmitter (UART)*. https://docs.arduino.cc/learn/communication/uart/. Accessed: 2025-04-28 (cit. on p. 15).

[21] hilscher. *SPI Interface*. https://www.hilscher.com/service-support/glossary/spi-interface. Accessed: 2025-05-25 (cit. on pp. 13, 20).

[23] H. Kagermann, W. Wahlster, and J. Helbig. "Recommendations for implementing the strategic initiative INDUSTRIE 4.0". In: *Final report of the Industrie 4.0 Working Group* (2013). URL: https://www.acatech.de/wp-content/uploads/2018/03/Final_report__Industrie_4.0_accessible.pdf (cit. on p. 1).

[24] 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. 13, 20).

[25] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. 8th. Hoboken, NJ: Pearson, 2021 (cit. on p. 16).

[26] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Second. MIT Press, 2017. ISBN: 978-0262533812 (cit. on p. 1).

[27] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[28] F. Moutinho and L. Gomes. "Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems". In: *Design and Architectures for Signal and Image Processing (DASIP)*. Ed. by A. Chakraborty and R. P. Sharma. Vol. 372. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2012, pp. 143–150. ISBN: 978-3-642-28254-6. DOI: 10.1007/978-3-642-28255-3_16 (cit. on pp. 10, 11, 18, 19, 24).

[29] *MQTT Version 5.0*. Tech. rep. Accessed September 2025. OASIS Standard, 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html (cit. on p. 17).

[30] 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. 2, 5, 6).

[31] 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. 8).

[3] J. Paulo Barros et al. "FROM PETRI NETS TO EXECUTABLE SYSTEMS: AN ENVIRONMENT FOR CODE GENERATION AND ANALYSIS". In: *Proceedings of the First International Conference on Informatics in Control, Automation and Robotics - Volume 2: ICINCO*. INSTICC. SciTePress, 2004, pp. 464–467. ISBN: 972-8865-12-0. DOI: 10.5220/0001142104640467 (cit. on pp. 8, 24).

[32] 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. 7, 22, 23).

[33] 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, 5).

[34] 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. 2024, pp. 17–24. URL: https://www.proquest.com/scholarly-journals/on-flexible-manufacturing-system-controller/docview/3119238675/se-2 (cit. on pp. 12, 20).

[35] J. Postel. *Transmission Control Protocol*. RFC 793. Internet Engineering Task Force, 1981-09. URL: https://www.ietf.org/rfc/rfc793.txt (cit. on p. 16).

[36] 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. 16).

[37] S. Rao and A. Fuller. *Low Voltage Translation For SPI, UART, RGMII, JTAG Interfaces*. Tech. rep. 2021. URL: www.ti.com (cit. on pp. 14, 15, 20).

[38] 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. 6).

[39] 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. 6).

[40] M. Singh and M. Rajan. "MQTT-SN: A Publish/Subscribe Protocol For Wireless Sensor Networks". In: *International Journal of Computer Applications* 119.12 (2015), pp. 1–5. DOI: 10.5120/21034-3973 (cit. on p. 17).

[41] I. Sommerville. *Software Engineering*. 9th. Pearson, 2011. ISBN: 978-0137035151 (cit. on p. 2).

[42] T. Stahl, M. Völter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN: 978-0470025703 (cit. on p. 2).

[43] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley Professional, 1994 (cit. on p. 16).

[44] 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. 9).

[45] 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. 16).

[46] J. F. Wakerly. *Digital Design: Principles and Practices*. 4th. Upper Saddle River, NJ: Pearson Prentice Hall, 2006 (cit. on p. 15).

[47] 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. 12, 20).

[48] Wikipedia. *Petri net*. https://en.wikipedia.org/wiki/Petri_net. Accessed: 2025-04-25 (cit. on p. 5).

[49] Wikipedia. *Universal asynchronous receiver-transmitter*. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter. Accessed: 2025-04-28 (cit. on pp. 14, 20).

[50] L. Zhang, W. Wang, and L. Sun. "An Implementation of MQTT-SN for Wireless Sensor Networks". In: *2013 IEEE International Conference on Green Computing and Communications*. IEEE, 2013, pp. 471–474. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.86 (cit. on p. 17).

A

Generated Code Snippets for I²C Protocol Example

Listing A.1: I²C Example Code

```
// ================================================================
// Code for: i2c Protocol
// Generated by IOPT API for your 'DemoProject' project.
// All generated code is intended for the net_io file.
// 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
// ================================================================

// ================================================================
// Code for Slave controller
// ================================================================

/*
* PART 1: Place this at the top of your file with the other global variables and includes.
*/
#include <Wire.h>
#define I2C_SLAVE_ADDRESS_for_buttonPress 9 // Address for this device
#define CMD_TRIGGER_buttonPress 'P' // Command to activate the event
volatile bool buttonPress_trigger_flag = 0;
void receiveI2CEvent(int byteCount); // Function prototype


/*
* PART 2: Place this inside your 'DemoProject_InitializeIO()' function.
*/
// Start the I2C bus as a slave with the defined address
Wire.begin(I2C_SLAVE_ADDRESS_for_buttonPress);
// Register the callback function that will be called when receiving data
Wire.onReceive(receiveI2CEvent);


/*
```

```
* PART 3: Place this inside your 'DemoProject_GetInputSignals()' function.
*/
if (buttonPress_trigger_flag == 1) {
events->buttonPress = 1;
buttonPress_trigger_flag = 0; // Reset the flag to fire only once
} else {
events->buttonPress = 0; // Ensure the event is inactive by default
}


/*
* PART 4: Place this at the bottom of your file with the other helper functions.
*/
void receiveI2CEvent(int byteCount) {
if (Wire.available() > 0) {
char command = Wire.read();
if (command == CMD_TRIGGER_buttonPress) {
buttonPress_trigger_flag = 1;
}
}
while (Wire.available() > 0) {
Wire.read();
}
}

// ================================================================
// Code for Master controller
// ================================================================


/*
* PART 1: Place this at the top of your file with the other global variables and includes.
*/
#include <Wire.h>
#define I2C_SLAVE_ADDRESS_for_buttonPress 9 // Address for this device
#define CMD_TRIGGER_buttonPress 'P' // Command to activate the event
void triggerMasterSendSlaveEvent(byte slaveAddress, byte command);


/*
* PART 2: Place this inside your 'DemoProject_InitializeIO()' function.
*/
while (!Serial); // Wait for Serial to be ready
Wire.begin(); // Start the I2C bus as a Master
Serial.println("I2C Master Initialized.");


/*
* PART 3: Place this inside your 'DemoProject_OutputSignals()' function.
*/
```

```
82 if (events->buttonPress == 1) {
83 triggerMasterSendSlaveEvent(I2C_SLAVE_ADDRESS_for_buttonPress,CMD_TRIGGER_buttonPress);
84 }
85
86
87
88 /*
89 * PART 4: Place this at the bottom of your file with the other helper functions.
90 */
91 void triggerMasterSendSlaveEvent(byte slaveAddress, byte command) {
92 Serial.println("Attempting to send command to slave...");
93
94 // Step 1: Begin a transmission to the I2C slave device
95 Wire.beginTransmission(slaveAddress);
96
97 // Step 2: Send the command byte
98 Wire.write(command);
99
100 // Step 3: Stop the transmission and send the data
101 byte error = Wire.endTransmission();
102
103 // Check the status of the transmission
104 if (error == 0) {
105 Serial.println("Command sent successfully!");
106 } else {
107 Serial.print("Error sending command. Error code: ");
108 Serial.println(error);
109 Serial.println("Check connections and slave address.");
110 }
111 }
```

B

Generated Code Snippets for UART Protocol Example

Listing B.1: UART Example Code

```
// ================================================================
// Code for: uart Protocol
// Generated by IOPT API for your 'SystemA' project.
// All generated code is intended for the net_io file.
// 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
// ================================================================




// ================================================================
// Code for Receiver Controller (listens for event: 'toggleLED')
// ================================================================

/*
* PART 1: Place this at the top of your file with the other global variables and includes.
*/
#include <HardwareSerial.h>

// -- UART Configuration --
HardwareSerial MySerial(2);
#define RXD2 16
#define TXD2 17

// -- Message and Flag Definitions --
const String message_uart_toggleLED = "trigger_toggleLED";
volatile bool toggleLED_trigger_flag = 0; // flag only for receiving signals

// -- Function Prototypes --
void setupUart_toggleLED();
String receiveDataUart();
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 == 1) {
43 events->toggleLED = 1;
44 toggleLED_trigger_flag = 0; // 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 = 1;
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 */
```

```
#include <HardwareSerial.h>

// -- UART Configuration --
HardwareSerial MySerial(2);
#define RXD2 17
#define TXD2 16

// -- Message Definitions --
const String message_uart_toggleLED = "trigger_toggleLED";

// -- Function Prototypes --
void setupUart_toggleLED();
void sendDataUart(String message);

/*
* PART 2: Place this inside your 'SystemA_InitializeIO()' function.
*/
setupUart_toggleLED();

/*
* PART 3: Place this inside your 'SystemA_PutOutputSignals()' function.
*/
if (events->toggleLED == 1) {
sendDataUart(message_uart_toggleLED);
events->toggleLED = 0; // IMPORTANT: Reset the event
}

/*
* PART 4: Place this at the bottom of your file with the other helper functions.
*/
void setupUart_toggleLED() {
MySerial.begin(9600, SERIAL_8N1, RXD2, TXD2);
Serial.println("UART communication initialized (Sender).");
}

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

C

Generated Code Snippets for TCP/MQTT Protocol Example

Listing C.1: TCP/MQTT Example Code

```
// ====================================================================
// Code for: tcp Protocol
// Generated by IOPT API for your 'SensorNetwork' project.
// All generated code is intended for the net_io file.
// 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
// ====================================================================

// ====================================================================
// Listener Controller (listens for event: 'alert')
// ====================================================================

/*
* PART 1: Place at the top with other globals
*/
#include <WiFi.h>
#include <PubSubClient.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

const char* mqtt_broker = "broker.hivemq.com";
const int mqtt_port = 1883;
const char* client_id = "ESP32_IOPT";
const char* topic_sub_alert = "sensors/events";

const char* message_tcp_alert = "trigger_alert";
volatile bool alert_trigger_flag = 0;

WiFiClient espClient;
PubSubClient client(espClient);

```

```
void tcpMqttInitializeIO();
void reconnect(const char* topic);
void loopDelayTcp(const char* topic);
void callback(char* topic, byte* payload, unsigned int length);

/*
* PART 2: Place inside SensorNetwork_InitializeIO()
*/
tcpMqttInitializeIO();

/*
* PART 3: Place inside SensorNetwork_GetInputSignals()
*/
if (alert_trigger_flag) {
events->alert = 1;
alert_trigger_flag = 0;
} else {
events->alert = 0;
}

/*
* PART 4: Place inside SensorNetwork_LoopDelay()
*/
loopDelayTcp(topic_sub_alert);


/*
* PART 5: Helper functions (place at the bottom)
*/
void tcpMqttInitializeIO() {
Serial.begin(115200);
WiFi.begin(ssid, password);
Serial.print("Connecting to WiFi...");
while (WiFi.status() != WL_CONNECTED) {
delay(500);
Serial.print(".");
}
Serial.println("\nWiFi connected!");
client.setServer(mqtt_broker, mqtt_port);
client.setCallback(callback);
}

void reconnect(const char* topic) {
while (!client.connected()) {
Serial.print("Connecting to MQTT broker...");
if (client.connect(client_id)) {
Serial.println("connected!");
client.subscribe(topic);
} else {
Serial.print("failed, rc=");
```

```
Serial.println(client.state());
delay(5000);
}
}
}

void loopDelayTcp(const char* topic) {
if (!client.connected()) {
reconnect(topic);
}
client.loop();
}

void callback(char* topic, byte* payload, unsigned int length) {
String msg = "";
for (int i=0; i<length; i++) msg += (char)payload[i];
msg.trim();
if (msg == String(message_tcp_alert)) {
alert_trigger_flag = 1;
Serial.println("Correct TCP message received. Flag activated for 'alert'.");
}
}


// ================================================================
// Publisher Controller (sends event: 'alert')
// ================================================================

/*
* PART 1: Place at the top with other globals
*/
#include <WiFi.h>
#include <PubSubClient.h>

const char* ssid_s = "yourNetworkName";
const char* password_s = "yourNetworkPassword";

const char* mqtt_broker_s = "broker.hivemq.com";
const int mqtt_port_s = 1883;
const char* client_id_s = "ESP32_IOPTSender";
const char* topic_pub_alert = "sensors/events";

const char* message_tcp_alert = "trigger_alert";

WiFiClient espClient_s;
PubSubClient client_s(espClient_s);

void tcpMqttInitializeIO_Sender();
void reconnectSender(const char* topic);
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()) {
```

```
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 }
```