POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Extending the capabilities of a virtual prototyping framework for functional and non-functional properties

Supervisors

Prof. Sara VINCO

Prof. Massimo PONCINO

Prof. Daniele Jahier PAGLIARI

Prof. Alessio BURRELLO

PhD. Giovanni POLLO

PhD. Mohamed Amine HAMDI

Candidate

Matteo ISOLDI

October 2024

Abstract

In recent years, the rapid advancements in Artificial Intelligence (AI) technologies and the increasing pervasiveness of Internet of Things (IoT) applications have made embedded platforms more complex to design and implement. Such systems are tightly coupled with the environment, that is continuously sensed to run the algorithms on fresh data, communicate the generated results to any external actors, and react accordingly through actuating features. This implies that, depending on the application, constraints that go beyond pure performance, such as response time, power consumption or safety, may be imposed, therefore requiring a greater amount of development time and resources to build the complete system. As a consequence, virtual prototyping solutions are steadily becoming more and more popular due to their ability to let designers explore multiple design variations and test them against the required specifications without (or before) building a hardware prototype. This allows also a drastic reduction in product costs and development time. Several simulators are already available on the market, targeting the simulation of specific properties of embedded systems. These properties can be divided into functional and extra-functional properties, where the former are related to the tasks that the system is expected to perform, while the latter refer to constraints on the manner in which the system implements and delivers its functionalities. This thesis focuses on the development of virtual platforms that include the monitoring of the extra-functional aspects. The cyber part (processor modeling and software simulation) is managed by a functional RISC-V instruction set simulator, GVSoC. This is integrated with the simulation of other peripherals and of power aspects (e.g., power consumption, power distribution policies and storage), that are implemented in SystemC-AMS. Both environments are C-based, and are part of a virtual platform called MESSY, constructed to run functional and power simulations in a single simulation run. To allow a more realistic simulation, it was necessary to open the platform to the communication with external tools, that would cover complex extra-functional aspects that would be too complex to model in a C-based approach. This has been addressed by implementing the capability to connect to external applications through a Unix socket component, named VirtualConnector. This module can be used by any simulated component contained within the tested system to exchange data during the simulation, allowing for more realistic scenarios to be created and validated. To demonstrate the potential introduced with the integration of the VirtualConnector, the virtual platform was connected to Webots, a popular robotic simulation software, to simulate the management of a robotic arm in a pick and place scenario. As an additional contribution, MESSY was extended with a more accurate implementation of the communication channel. Due

to its original high-level implementation, data exchanges between the core of the architecture and the other system components occurred instantaneously and were very limited. For these reasons, the channel was reworked to establish a low-level functional bus. The Advanced eXtensible Interface (AXI) protocol was chosen for the newly developed interconnection. It was demonstrated that the integrated channel introduces a negligible simulation overhead of around 2% compared to the original implementation.

Acknowledgements

Vorrei iniziare col ringraziare Amine e Giovanni, i migliori dottorandi che potessero capitarmi. Grazie alla loro esperienza e, soprattutto, alla loro enorme pazienza, è stato possibile concludere questo percorso universitario senza intoppi e con il sorriso. A seguire, vorrei ringraziare il pagliaccio con cui ho lavorato in questi otto lunghi mesi. Grazie per avermi sopportato, per tutto l'aiuto che mi hai dato e per aver trasformato i momenti di lavoro in ulteriori occasioni di divertimento. Grazie per aver riacceso la mia passione per Clash Royale ed avermi reso dipendente dai video spacchettamento del nostro Ciccio.

Vorrei poi ringraziare la combriccola Volpianese, composta dai migliori amici che si possano avere. I momenti trascorsi insieme sono e saranno sempre indimenticabili. Grazie per avermi supportato durante questo lungo percorso, per essermi stati vicini e per tutte le risate condivise. Un ringraziamento viene fatto anche agli amici pendolari, con i quali ho condiviso l'odio per il sistema dei trasporti pubblici, l'ansia e la paura prima di un esame, e la semplice stanchezza dopo una lunga giornata al Politecnico. Grazie a voi, prendere un treno non è mai stato così importante e divertente.

Successivamente, vorrei ringraziare l'eSports team di cui ho fatto parte durante la triennale. Vi ringrazio per avermi supportato ed essere stati parte del mio percorso, per tutte le chiamate e le centinaia di ore passate a rankare su Rocket. Grazie a voi, le ore di lavoro passavano più velocemente e pesavano molto meno sulle successive attività di studio. Inoltre, senza di voi, non penso che avrei vissuto quel periodo con tale spensieratezza. A seguire, vorrei ringraziare i membri della Pausa™ per il loro supporto ed incoraggiamento in tutti questi anni. Grazie per tutti quei momenti di gioia e meme che abbiamo potuto condividere.

Infine, vorrei ringraziare la mia famiglia, il cui supporto è stato fondamentale per concludere questo difficile percorso. Sarò per sempre grato per la vostra pazienza e l'incoraggiamento ricevuto nei momenti più bui. Senza di voi, non sarei mai riuscito a raggiungere questo traguardo.

Table of Contents

List of Tables

List of Figures

Acronyms

ACE	AXI Coherency Extension
AHB	Advanced High-performance Bus
AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Program Interface
AR	Read Address
AW	Write Address
AXI	Advanced eXtensible Interface
B	Write Response
CHI	Coherent Hub Interface
DMA	Direct Memory Access
DSE	Design Space Exploration
FCI	Franka Control Interface
FLOPS	FLoating point Operations Per Second

ILP	Instruction Level Parallelism
IoT	Internet of Things
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
JSON	JavaScript Object Notation
MCU	Microcontroller Unit
PULP	Parallel Ultra Low Power
R	Read Data
RISC	Reduced Instruction Set Computer
SLS	System-Level Simulation
SoC	System-on-a-Chip
TDF	Timed Data Flow
TLM	Transaction Level Modeling
URDF	Unified Robot Description Format
W	Write Data
XML	eXtensible Markup Language

Chapter 1

Introduction

In recent years, the increasing pervasiveness of Internet of Things (IoT) applications and the rapid advancements in Artificial Intelligence (AI) technologies have made embedded platforms more complex to design and implement. Such systems are tightly coupled with the environment, which is continuously measured and altered based on the decisions of the executed algorithms. This implies that, depending on the application, constraints that go beyond pure performance, such as response time, power consumption, reliability, quality of service, security or safety, may be imposed, therefore requiring a greater amount of development time and resources to build the whole system. As a consequence, virtual prototyping solutions are steadily becoming more and more popular due to their ability to let designers explore multiple design variations and test them against the required specifications without (or before) building a hardware prototype. This allows also a drastic reduction in product costs and development time. Several simulators are already available on the market, while many others are being developed, targeting the simulation of specific properties of embedded systems. These properties can be divided into functional and extra-functional (or non-functional) properties, where the former are related to the tasks that the system is expected to perform, while the latter refer to constraints on how the system implements and delivers its functionalities.

This thesis focuses on the development of a virtual platform to monitor the non-functional aspects of an integrated system. A functional RISC-V Instruction Set Simulator (ISS) called GVSoC is employed to model and simulate the core of the embedded platform. It is integrated with the simulation of other peripherals and power aspects, such as power consumption, distribution policies and energy storage within the system, which are implemented in SystemC-AMS. Both environments are C-based, and are part of a virtual platform called MESSY, constructed to run functional and power simulations in a single simulation instance. Specifically, the thesis work focused on improving the framework to enable more realistic and

comprehensive simulations. The most significant contribution of this work is the implementation of an internal component named VirtualConnector, which extends the platform with the capability to connect to external applications through a Unix socket. This allows other software systems to handle and manage more complex aspects of the system that would be difficult to model using a C-based approach. Such an addition also enables a more realistic approach to testing and validating the target platform, allowing the data generated by peripherals to vary, thus creating new inputs for the algorithms being executed and the decisions affecting the nearby environment. To demonstrate the newly integrated connectivity, the virtual platform was connected to Webots, a popular robotic simulation software, and tasked with managing the movements of a Franka Emika Panda manipulator. The arm, equipped with a camera sensor, produces images that are analyzed by an object detection network running on a RISC-V processor to determine the location of the target block. Once the coordinates have been determined and sent to Webots, the manipulator moves to the indicated location, ready to pick up the object.

As an additional contribution, MESSY was extended with a more accurate implementation of the communication channel. This enhancement was made to allow data exchanges between the core of the architecture and other system components to occur more realistically, taking the correct amount of time. The interconnection was reworked to establish a real, protocol-based, low-level functional bus. The Advanced eXtensible Interface (AXI) protocol was chosen for the newly developed interconnection due to its simplicity and popularity in the embedded domain. However, the protocol was not fully implemented; instead, it was integrated in a form suitable for the architecture, avoiding the addition of structures and procedures that are not used and would hinder its internal processes, such as arbitration mechanism, protection level support or atomic accesses. The newly implemented bus channel was tested using the same pick and place application that was used to validate the VirtualConnector module. It was demonstrated that the AXI-based interconnection introduces a negligible simulation overhead of around 2% compared to the original implementation of the functional bus.

This publication is organized into six chapters, with the first one introducing the challenges addressed by the thesis work. Chapter 2 introduces and explores the theoretical knowledge underlying the thesis, ranging from the notion of System-Level Simulation (SLS) to the world of robotics, and finally completing the background with descriptions of sockets and bus protocols. In Chapter 3, the tools employed during the thesis work are presented, along with an analysis of the current state of knowledge around the explored topics. The motivations behind the decisions made and the descriptions of the implementation processes, instead, are presented in Chapter 4. Within this chapter, the major contributions of the thesis are highlighted:

- the development and integration within MESSY of a C-based library named

VirtualConnector, which allows the simulation platform to connect to external software systems in order to extend its capabilities;

- the creation of a simulation scenario involving the handling of a robotic manipulator with specific movements' contraints;
- the development and implementation within the simulation framework of a low-level, AXI-based, communication channel.

Finally, Chapter 5 presents the insights obtained from analyzing and evaluating the framework's performance, while Chapter 6 contains reflections on the completed project, along with ideas for further extending the work.

Chapter 2

Background

This chapter aims at introducing all topics that are necessary for the full comprehension of this thesis. It begins with an explanation of the underlying concept of the simulators that have been used during this work (see Section 2.1). Next, it delves into a comprehensive description of the RISC-V standard (see Section 2.2) and a thorough explanation of robotic systems (see Section 2.3). To conclude, the chapter presents basic information regarding sockets (see Section 2.4) and bus protocols (see Section 2.5), highlighting the tools and methodologies chosen during the implementation work.

2.1 System simulation for virtual prototyping

A System-Level Simulation (SLS) is a set of tools and techniques that are used to simulate the behaviour of cyber-physical systems, entities composed of hardware components regulated by software programs. Due to the complex nature of these systems, this kind of solutions are more frequently used during their development cycle to explore multiple design variations, test them against the requested specifications and tune them accordingly, before actually building the hardware prototype, making them more common as **virtual prototyping** methods.

Recently, the utilization rate of virtual prototyping solutions is steadily increasing, thanks to the resulting drop of both development's time and cost for integrated systems. Many simulators are already available on the market or are being developed, each with its own set of characteristics. Some of the most important features to keep in mind when researching such systems are the number and type of properties of the target system that can be correctly simulated. Properties can be separated into two main categories: **functional** properties and **non-functional** (or extra-functional) properties. The former are related to the tasks that the system is expected to perform, while the latter refer to constraints on the manner in which

the system implements and delivers its functionalities. For cyber-physical systems, important extra-functional properties are power consumption, thermal modeling, fault-tolerance, reliability, efficiency and security. In the next chapter, different simulators will be examined in detail: an extra-functional simulator for evaluating power performances of embedded platforms, named MESSY [1], a functional Instruction Set Simulator (ISS) for a RISC-V based platform, named GVSoC [2], and a functional simulator for robotic systems, called Webots [3].

2.2 RISC-V

Born in May 2010 as part of the Parallel Computing Laboratory at UC Berkeley, RISC-V is an open-source Instruction Set Architecture (ISA). Due to being an open standard, anyone can access and implement it free of charge, avoiding costly licensing fees, while, at the same time, tailoring it to their target end applications with respect to power, performance and area (PPA) metrics. Its royalty-free nature and great customization options lead to a surge in interest and adoption across the industry, which, in turn, collaborates with the foundation maintaining the project, the RISC-V Foundation, to further evolve the architecture.

The RISC-V architecture is built upon a set of key design principles that contribute to its performance, efficiency and adaptability. First, the instruction set is based on a load-store, **Reduced Instruction Set Computer (RISC) architecture**, designed to simplify the individual instructions through a small, fixed and standardized length. This allows the hardware implementation of the processor to be simpler, less consuming and more efficient. Furthermore, this facilitates the implementation of Instruction Level Parallelism (ILP). Second, the ISA is designed to be **modular** and easily **extensible**. A RISC-V based processor is built starting from independent components that can be combined together in various ways to attain the necessary capabilities to perform its tasks. Extensions, instead, allow to add new features or instructions without disrupting existing functionalities. The base of the instruction set specifies integer instructions, control flow, registers, memory and addressing logic to implement the core of a general-purpose computer, whereas extensions provide domain specific logic, such as floating-point arithmetic, vector processing, and cryptographic operations. Denoted with a single, capital letter, the most notable standard extensions are: **M**, **F** and **D**. M adds support for integer multiplication and division instructions. The F and D extensions, instead, add support for Single-Precision and Double-Precision floating point operations.

The RISC-V base instruction set is comprised of four instruction formats called *R-type*, *I-type*, *S-type* and *U-type*. As can be seen in Figure 2.1, both the sources (*rs1* and *rs2*) and the destination (*rd*) are kept int the same position, therefore permitting a simplification of the decoding hardware. For the same reason the

immediates were placed towards the leftmost significant bits and the sign bit position is always the most significant bit (MSB) of the instruction.

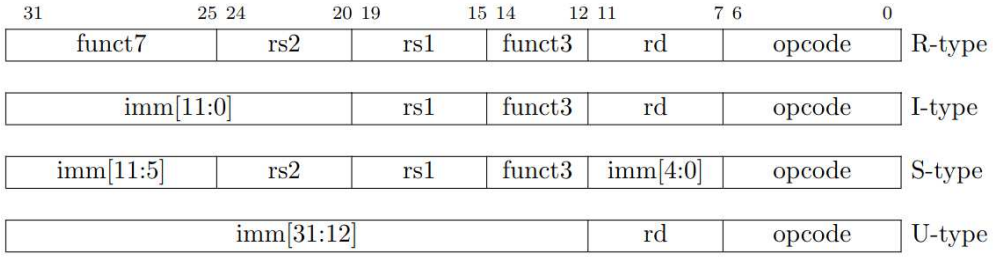


Figure 2.1: RISC-V instruction formats [4]

Since its creation, numerous cores and SoCs implementations have been proposed from members of the industry and research institutions, like CV32E40P [5], NEORV32 [6] and Rocket Core [7], while various articles have been published to review and compare them across different jobs [8, 9]. One of the major platforms used in the embedded and IoT domains is the Parallel Ultra Low Power (PULP) Platform [10], created as a collaborative effort between ETH Zurich and the University of Bologna. This architecture targets advanced integrated systems that need to process data streams generated by multiple sensors with the goal of increasing their energy efficiency. This is achieved through clusters of RISC-V cores that share a tightly-coupled data memory. To also aim at more simpler systems, two additional platforms were designed based on PULP: PULPissimo [11] and PULPino [12]. These two contain only a single RISC-V core and, for this reason, the memory and cache sharing infrastructure is not present.

2.3 Robotic systems

The term robotic system is used to refer to an integrated system of computer-controlled manipulators, designed to perform tasks semi-autonomously or autonomously. More commonly called **robots**, their goal is to aid and assist humans with repetitive, unpleasant, complex or hazardous jobs, like performing high-precision surgical operations [13] or exploring unstable ruins to find survivors after an environmental hazard occurred [14, 15]. These systems are based on a mechanical frame and contain several electrical components that are controlled by a software program, executed directly on the robot itself. Various parts play a key role in this architecture: a microcontroller unit (MCU) or a system-on-a-chip (SoC) is used to control the entire robot, while sensors and actuators are used to perceive information from the nearby environment and react to ever changing conditions. The number of joints, as well as their type, are essential to determine

the movements and tasks that can be achieved with a certain robot structure, whereas the power source defines the source of energy that will aid the system fulfill its needs. Lastly, an important factor is the software that is going to be executed by the robot. In fact, depending on its quality and refinement, efficiency and performance can be greatly affected. Therefore, many manufacturers tend to provide high-level interfaces to program the robot and virtual models of its structure to easily prototype its trajectories inside a virtual environment. In the following sections, some of these components will be described in more detail.

There exist many types of robots, each with its own target environment, use case and properties. For this reason, it is difficult to classify them, due to the many parameters to account for. The most popular categorization separates robots based on their application. Following this organization, a robot can be labeled as:

- a **modular robot**, which is a machine built upon a modular architecture. Their design allows for identical modules to be put together or removed from the overall system, therefore creating a more complex and flexible structure, capable of adapting to the circumstances and tasks at hand. Due to their higher degree of freedom, though, they are quite challenging to develop, which is why they are still mostly a research topic. Two of the most recent and refined works are ReBiS [16], a robot that can switch between snake-like and bipedal motions, and AuxBots [17], modular units with high expansion and large force capabilities;

- an **educational robot**, created to assist teachers during classes. Renowned examples of such robots are the Turtle robots from Valiant Technology[1] and the Lego Mindstorms robotic kits released in 2006 by Lego;

- a **mobile robot**, capable of moving, independently or through special control devices, inside an uncontrolled environment. Very common in commercial and industrial settings, they are typically used to move materials efficiently, even though the most famous examples of such category are the vacuum cleaner robots realized by iRobot[2], referred to as Roombas;

- an **industrial robot**, which is employed to automate heavy production jobs in industrial settings, reducing costs while, at the same time, enhancing accuracy, reliability and quality of the process or product it contributes to. Due to their tasks, their characteristics can vary a lot, therefore as a general definition, the one provided by the ISO 8373:2021 standard is typically used;

[1] `https://roamerrobot.tumblr.com/post/23079345849`

[2] `https://www.irobot.com/en_US/roomba.html`

- a **cobot**, or collaborative robot, built to work safely alongside human workers in a shared, collaborative workspace. They represent a subset of the industrial robots, since they are usually charged with simple, repetitive tasks to complement the more complex and thought-intensive duties assigned to workers. Such systems can be found during the welding, gluing, polishing and packaging processes of a product's production [18], but they can also be used to tend to other machines or to help speed up secondary processes, like the pick and place manipulators [19], which help moving workpieces around the location.

In recent years, this last category has been the subject of an explosive growth, mainly due to the lower cost, smaller size and high return of investment of such machines in small to medium-sized environments. Nowadays, the most widely employed systems are produced by ABB Robotics, FANUC Corporation and Universal Robots. A company that is recently experiencing a growth in this sector is Franka Robotics, an organization based in Germany that provides robotic arms integrated with AI technologies. At the end of this section, a brief description of the cobot that has been employed during the thesis work, the **Franka Emika Panda**, is provided.

2.3.1 Microcontroller units

A Microcontroller Unit (shortened to MCU or μC) is a complete system built upon a single board or chip. It typically contains a processor that can provide low computational power, some programmable I/O peripherals, a small main memory module and, in some cases, a Flash storage. A block diagram of a general microcontroller is represented in Figure 2.2.

Due to their modest capabilities, they are used to perform small and repetitive tasks for application-specific embedded systems, usually involving their sensing and actuating peripherals. In the field of robotics, their small size and inexpensiveness make them essential units for controlling the motions of the manipulators, as these programs are not computationally intensive.

2.3.2 Sensors and actuators

These two represent the extremes of an embedded device's chain of operations, which is summarized in Figure 2.3. Sensors measure physical quantities from the external world and converts them into data that can be interpreted by a digital processor, while actuators are used to alter the nearby environment as a response to certain conditions.

[3] `https://www.playembedded.org/blog/microcontrollers-101/`

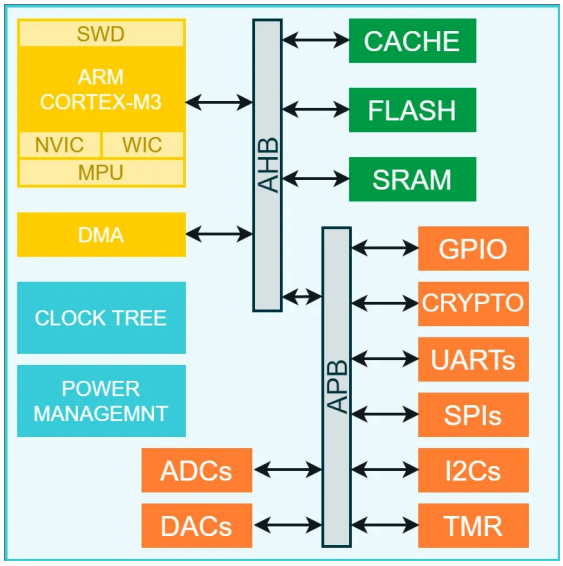


Figure 2.2: General scheme of a microcontroller[3]

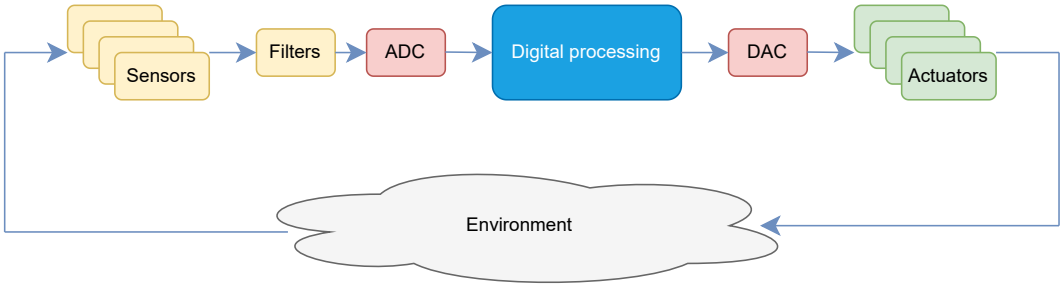


Figure 2.3: Dataflow of an embdedded device

As part of a robotic system, sensors are generally used to determine when specific actions have to executed. Take for example a Roomba robot: some of its sensors are of the infra-red type and are used to determine whether an obstacle is in the vicinity and if it will impact its movements. Actuators, on the contrary, are used to execute those actions. Robots have plenty of actuators, mostly used to move its structure around. As an example, think of a robotic arm: these usually have two or more axis, each related to a certain motor, which can move the manipulator in a specific direction, within predefined limits.

2.3.3 Control and automation

Robotic systems can have different levels of autonomy, ranging from being directly controlled by human workers to being totally independent for extended periods

of time. Anytime some level of autonomy is expected from the machine, whether to accomplish simple jobs or high-level tasks, the complexity of its processing capabilities increases. These are essential due to their significance in evaluating the data acquired from the sensors and determining the most adequate response, while taking into consideration the environment, the mechanical form of the robot, its movements' constraints and how the interactions can be conducted. In recent years, in order to improve these capabilities, cognitive models, mapping and motion planning techniques are trained and implemented through AI models [20]. Together with pattern recognition and computer vision models, these allow the robot to analyze its circumstances, determine the relative positions of targets and obstacles and move accordingly without any kind of human involvement [21].

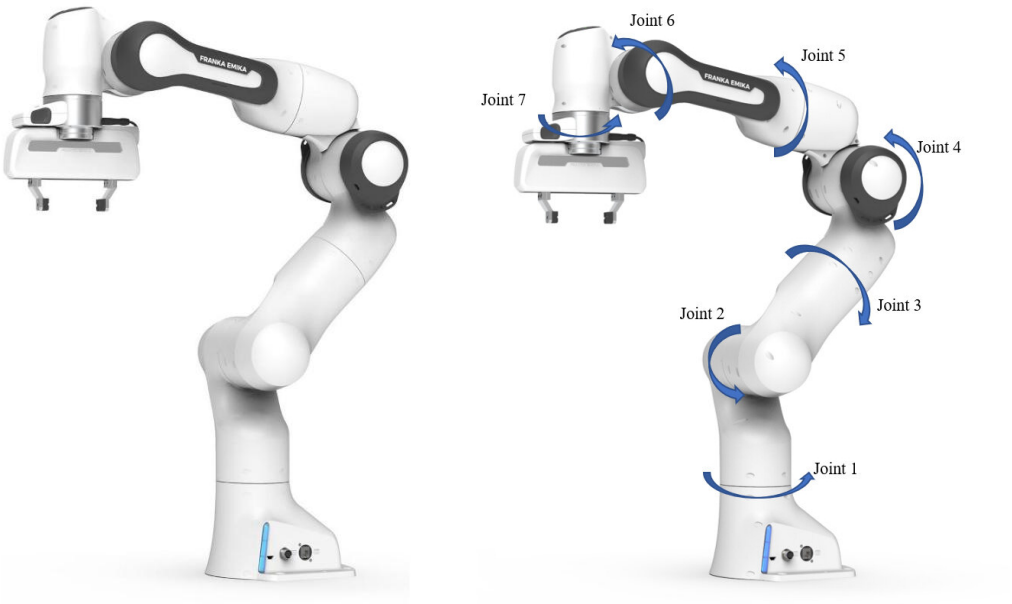
To help customers set up and examine the system's behaviour in working circumstances, virtual prototypes and machine-learning models to train are provided by manufacturers, allowing the usage of simulators before making use of the physical hardware. In case specific movements have to be manually planned, to properly handle critical situations, kinematics and dynamics libraries exist for many languages and very often support many well-known robot models. During the thesis work, one such library, called Robotics Toolbox for Python [22], has been included to handle the Emika Panda's trajectories in Webots, a process detailed in Section 4.1.3. In particular, it has been used to determine the final joints' positions, starting from the target space coordinates. This is a common problem in robotics known as **inverse kinematics**.

2.3.4 Franka Emika Panda

The Emika Panda, shown in Figure 2.4, is a 7-axis robotic arm, capable of up to $850mm$ of reach. Currently superseded by the Production 3 and Research 3 manipulators of the same brand, this robotic arm was one of the first attempt at designing collaborative robots for small businesses, with the goal of making the technology much more accessible. Together with the smart "sense" of touch of its end-effector, the Franka Hand, the easy-to-use software provided to program the robot's movements and the amount of freedom granted by the seven axis, it earned a lot of recognition, including being appointed as the "Best Invention" by TIME Magazine in 2018[4].

An additional factor that granted the arm significant fame in the industry is the possibility to prototype its routines inside a virtual environment. Franka Robotics provided models of this arm for every major robotic simulation software, like Webots and the Robotic Operating System (ROS). Together with the Franka

[4] `https://time.com/collection/best-inventions-2018/5454734/panda/`



(a) The Emika Panda cobot [23] **(b)** The movement of the joints highlighted [24]

Figure 2.4: The Franka Emika Panda robot manipulator

Control Interface (FCI), a library that allows to establish a direct communication channel with the arm, developers could create a controlled and reproducible testing environment that could be transferred to the manipulator at any given moment, thus improving the development cycle's efficiency while reducing the overall production costs.

2.4 Network sockets

Sockets are software structures that allow different processes, which may run on the same machine or reside across the network, to exchange data. They are one of the many mechanisms provided by the operating system to grant Inter-Process Communication (IPC) and, thanks to their easy-to-use nature and minimal performance overhead, they are typically used for client-server applications. The lifetime of a socket is limited to that of the process that created it, while its properties are defined by the Application Program Interface (API) of the network architecture it is part of. All of the information about a socket are stored by the operating system within a file descriptor.

Sockets can be identified as **IP sockets** or **Unix sockets**. IP sockets, generally known as TCP/IP sockets, are used whenever communication has to occur over the

network. Unix sockets, conversely, are used for transmissions between processes running on the same machine. The former can also serve this scenario, but Unix sockets are more efficient in this regard, since they are designed to not rely on networking structures and protocols for the exchange of messages. A further subdivision of sockets is based on the type of service provided:

- **datagram sockets** provide a connection-less service. Packets are routed independently and reliability is not guaranteed. This means that messages can arrive out of order or, in the worst case, can be missed;
- **stream sockets** are based on a connection-oriented approach. The TCP protocol is used for data transfer, making transmissions more secure and reliable. In case a message doesn't reach its destination, a request for the lost packet is sent back.

2.4.1 Sockets in a client-server architecture

In a client-server architecture, sockets are used differently. The server node, which exposes a service, creates a socket to handle all client requests. During its creation, basic information such as its type (IP or Unix) and kind of service quality required (datagram or stream) need to be specified. Once the socket is created, it needs to be bound to an address or file descriptor in order to be seen by other processes. After binding the socket, communications can finally occur. By using the appropriate API function, the socket starts to wait for incoming connections and, after accepting them, data can be read from or written to the client nodes until the connections are closed. From the client's point of view, instead, the socket can be used right after its creation, without having to bind it to any address or file descriptor. To establish a connection, the only information required is the address of the server's endpoint. After the creation of the connection, data transfers occur until the channel is closed. The whole process for Unix sockets is documented in Figure 2.5.

2.5 Bus protocols

A bus is a system that connects different components inside a computer, allowing them to exchange data, power, and control signals to communicate and cooperate. To manage the various transmissions that can occur on the channel, a set of rules and conventions, known as a **bus protocol**, have to be followed by the interacting entities. The protocol is essential, since it defines the guidelines and norms for addressing, synchronisation, and data transfer, such as how data should

[5] `https://www.educative.io/answers/unix-socket-programming-in-c`

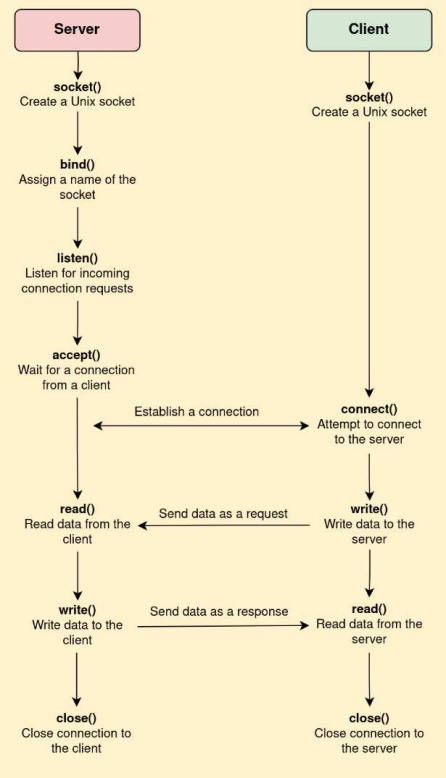


Figure 2.5: Sockets in a client-server architecture[5]

be transmitted, the timing of the signals of each transmission, how to regulate concurrent requests on the bus (bus arbitration), the error detection and correction mechanisms and many other details. Thus, a bus protocol ensures that the communications are reliable, efficient and consistent.

One of the most widely used bus protocols in embedded systems is the **Advanced Microcontroller Bus Architecture** (AMBA), an open-standard developed by Arm for high-performance, on-chip communications in microcontroller systems. The standard comprises five protocols:

- the **Advanced High-performance Bus** (AHB), responsible for high performance interconnections between high-speed peripherals, processors, and memory units;
- the **Advanced Peripheral Bus** (APB), which focuses on efficient and low-bandwidth connectivity with peripherals' register interfaces. It proposes a simple address and data phase with a low complexity signal list;
- the **Advanced eXtensible Interface** (AXI), an advanced system bus that supports multiple masters and slaves, burst transfers, out-of-order transactions

and quality of service for high-performance interconnects;

- the **AXI Coherency Extension** (ACE), an extension of AXI that supports system-wide cache coherency. At the same time, it also enables one-way coherency, allowing network interfaces to read from the caches of a fully coherent ACE processor;
- the **Coherent Hub Interface** (CHI), an high performance system bus with additional features to manage traffic congestion.

The following section will focus on AXI, the protocol that has been integrated within MESSY during the thesis work. Its implementation on the extra-functional simulator, instead, will be described in a later chapter.

2.5.1 The AXI protocol

Designed to target high performance and high clock frequency systems, Advanced eXtensible Interface (AXI), introduced with the third version of the AMBA protocol, is an interface specification. Thus, rather than on the bus, its focus is on the **interfaces** that need to be integrated by each module of the system. Two types of interfaces are defined by the standard, which are involved in every transmission: **manager** and **subordinate**. These are symmetrical and, therefore, contain the same signals, facilitating their integration.

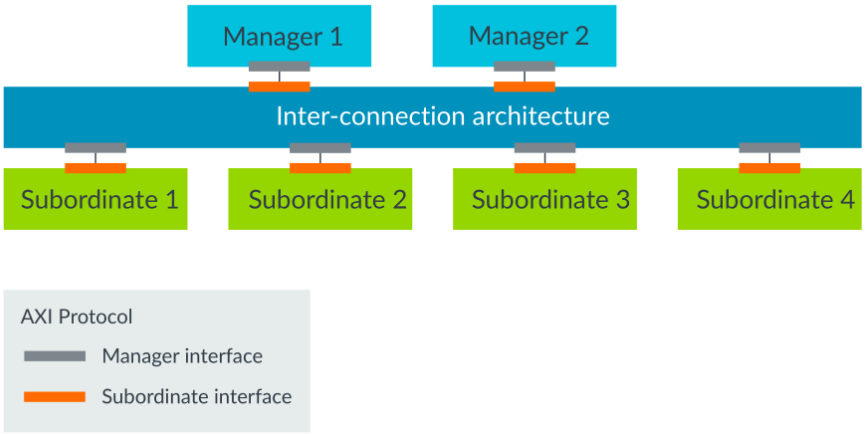


Figure 2.6: AXI multi-master interconnection [25]

The AXI specification describes a point-to-point protocol between the two interfaces. Transactions are always started by a manager module and they target a single subordinate block at a time. Figure 2.6 shows the interfaces arrangement

for a two-manager, four-subordinate system. Note that the interconnection also implements them: this is caused by the point-to-point nature of the standard. Additionally, due to the protocol design, the bus has to act as a routing component for each exchange, though its implementation is left to the designers. The interfaces use for communication five different channels, as evidenced by Figure 2.7. Write Address (AW), Write Data (W) and Write Response (B) are used for manager to subordinate write operations, while Read Address (AR) and Read Data (R) are used for reading data from subordinate blocks. Having separate channels for the two operations helps to improve bandwidth performances of the interfaces. In the

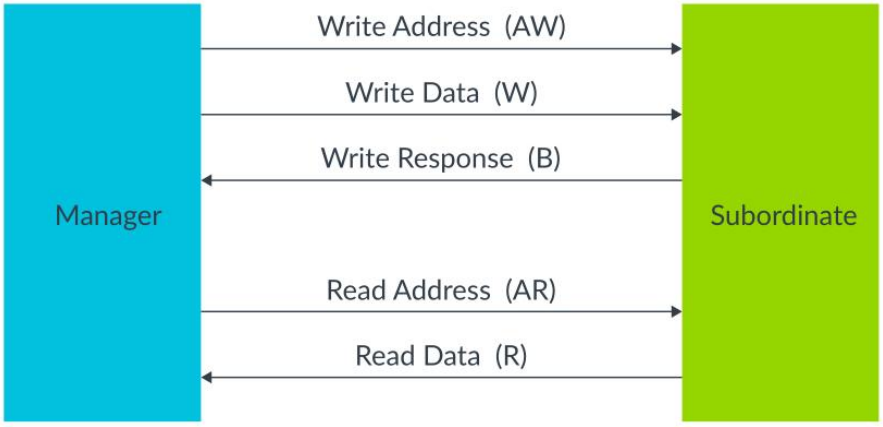


Figure 2.7: AXI channels [25]

following, a simple explanation of how a write and read operations are executed is given, considering only the signals and procedures implemented during the thesis work (see Section 4.2.1 for more details on these restrictions). A list of all the signals implemented is reported in Table 2.1.

AW channel	**W channel**	**B channel**	**AR channel**	**R channel**
AWADDRESS	WDATA	BRESP	ARADDRESS	RDATA
AWVALID	WVALID	BVALID	ARVALID	RVALID
AWREADY	WREADY	BREADY	ARREADY	RREADY
	WLAST		ARLEN	RRESP
				RLAST

Table 2.1: Signals implemented during the thesis work for each channel

A write operation is composed of three phases. The first one takes place on the AW channel. Here, the manager block writes the address of the first register it needs to update on the *AWADDRESS* channel and signals the beginning of a transmission

to the other system blocks by raising the *AWVALID* line. Then, it waits until the corresponding subordinate block raises the *AWREADY* bit, indicating that the transaction can safely occur. Once both *AWVALID* and *AWREADY* have been raised, the communication moves on to the second phase, occurring on the W channel. At this time, the subordinate is waiting for data with *WREADY* set to high. When the manager is ready, it starts sending data on the *WDATA* data line. To indicate that the messages are being sent, the *WVALID* line is raised with each exchange. Upon the last data, the *WLAST* bitline is raised, to indicate to the subordinate that, after storing the current message, it will need to move to the final phase. During this last phase, occurring on the B channel, the manager is on hold with *BREADY* high, while the subordinate examines the received messages and produces a response to send back. This response is sent through *BRESP* with the *BVALID* bit set to high at the same time. After receiving the response, the transmission is over. A graphical representation of the process can be found in Figure 2.8.

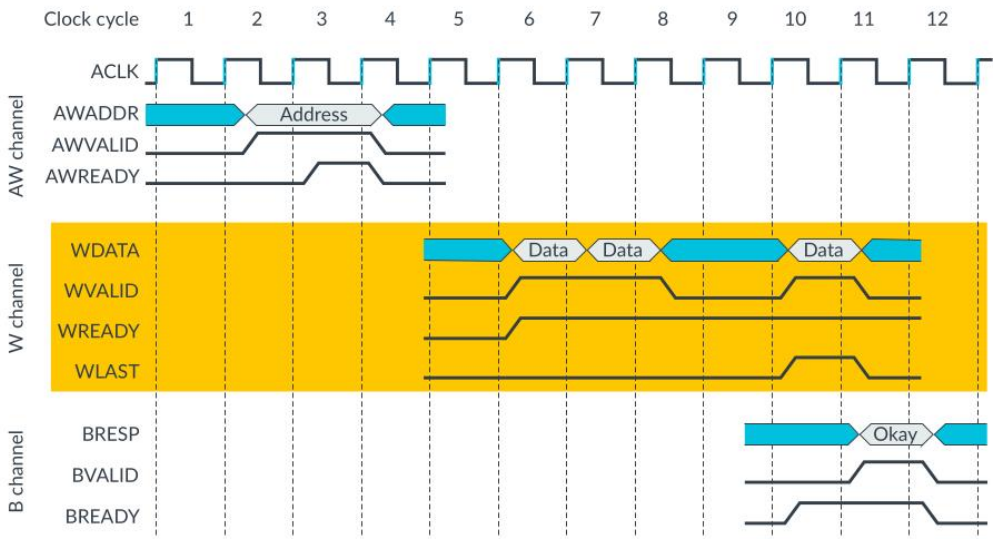


Figure 2.8: AXI write transaction [25]

A read operation, instead, is composed of two phases. The first one takes place on the AR channel. Here, the manager block writes the address of the first register it needs to read on the *ARADDRESS* channel and signals the beginning of a transmission to the other system blocks by raising the *ARVALID* line. Then, it waits until the corresponding subordinate block raises the *ARREADY* bit, similarly to what happens during the first phase of a write operation. An additional signal, *ARLEN*, is used to define how many registers the master node wants to read from the subordinate. Once both *ARVALID* and *ARREADY* have been raised, the

transaction can move to the second phase on the R channel. Contrary to the write operation's second phase, in this situation it is the subordinate that takes control of the data and valid signals, while the manager is waiting with *RREADY* high. For each data transmitted, the subordinate needs to have the *RVALID* line raised, while updating *RDATA* and, when the last element to send is reached, *RLAST*. An ulterior difference with the write operation is that the response is sent together with the data, thus requiring the subordinate interface to also update the *RRESP* data line during each exchange. Once all data have been transmitted, the communication is over. The process is shown in Figure 2.9.

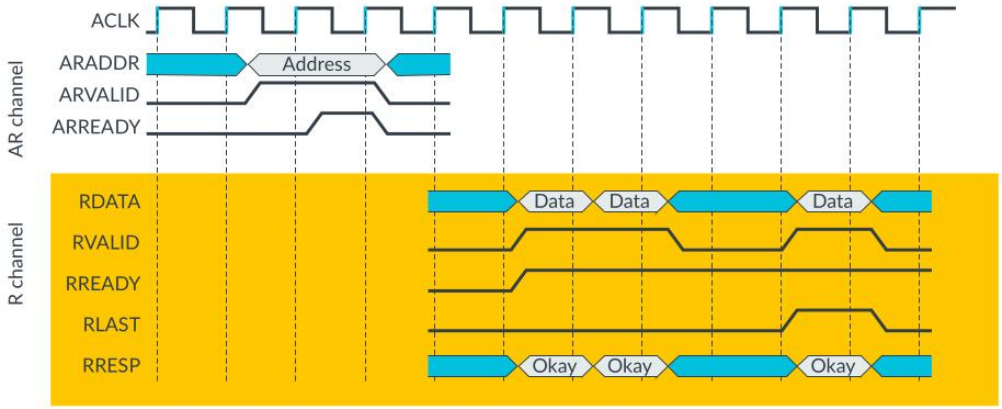


Figure 2.9: AXI read transaction [25]

Chapter 3

Related works

This chapter aims at presenting the most relevant works for this thesis and the current state of knowledge around the explored topics. First, it delves into a thorough description of the simulators constituting the foundation of this work: GVSoC (see Section 3.1), a functional ISS for RISC-V based processors, MESSY (see Section 3.2), a simulation framework for non-functional properties, and Webots (see Section 3.3), a robotic simulation tool. Afterwards, it briefly explores attempts at creating a communication channel between GVSoC and Webots and functioning AXI implementations for SystemC-based systems (see Section 3.4).

3.1 GVSoC: an event-driven simulation platform for PULP-based architectures

GVSoC [2] is an open-source simulator for PULP-based architectures. It can simulate complex platforms, which can include multiple cores, multiple memory levels, multiple I/O peripherals, and accelerators, with less than 10% of error with respect to the physical hardware implementation. This level of accuracy is reached while simultaneously providing a faster simulation experience compared to cycle-accurate simulators. As a consequence, the design space can be explored more easily and quicker, facilitating developers during the early stages of system development. Another key factor that enables faster DSE through GVSoC is given by its structure. The simulator, in fact, is based on three key components: C++ models, used to describe the behaviour of system components such as cores, memories and peripherals; configuration JSON files to customize the parameters of the instantiated modules; and a set of Python generators to instantiate all components of the system. This modular structure allows to compile the C++ models of the system components at the beginning and then to build a specific platform modifying the JSON files without recompiling the simulator.

GVSoC simulation is event-driven. All events are stored in a circular buffer, which is read by the simulation engine to determine the next actions to process. To include the simulator in a more complex system, GVSoC exposes some APIs to interact with the simulation time and event queue. One example of such functions is the *step_until()* method, which advances the simulation until the given timeframe and returns the timestamp of the next event of the queue.

During the thesis work, GVSoC was used to simulate an implementation of a PULP-based microprocessor called GAP8[1], developed by GreenWaves Technologies.

3.2 MESSY: a system-level simulation framework for extra-functional properties

MESSY (Multi-layer Extra-functional Simulator in SystemC) [1] is an open-source framework that integrates a functional RISC-V simulator, namely GVSoC, with SystemC-AMS to model extra-functional aspects, in detail power storage and distribution. To manage this integration, a bus-centric paradigm is exploited, where an interconnection is modeled for each aspect to be simulated. Each system component is then connected to each bus by implementing one model per interconnection. In this way, the functional model of a module handles its behaviour, while the power model determines the estimates of power demand. Figure 3.1 shows the architecture of the framework. The combination of both functional and extra-functional aspects in a single simulation framework allows to perform a Design Space Exploration (DSE) that accounts for the mutual impact between them, providing a significantly improved design experience.

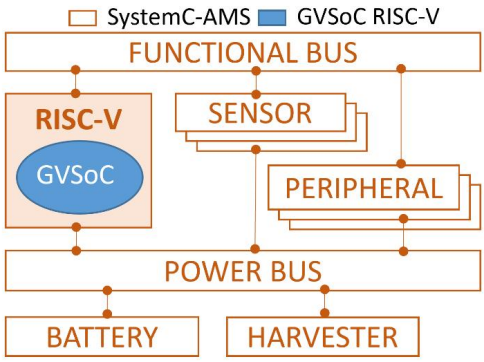


Figure 3.1: High-level scheme of the framework architecture [1]

[1] https://greenwaves-technologies.com/gap8_mcu_ai/

MESSY, in its current state, is shipped with GVSoC as its functional simulator. However, the modularity of its architecture allows it to connect it to any functional ISS that satisfies certain constraints:

- the simulator must be written in C/C++;
- the simulator must model instruction-level timing, to allow MESSY to make estimates on power consumption over time;
- the simulator must be embeddable as a software component within a SystemC module and must expose APIs to export state information or direct power estimates.

For the development of this thesis, MESSY was chosen for its great modularity. In fact, the framework allows the definition of the list of system components through a JavaScript Object Notation (JSON) file. Here, customization parameters can be defined for each component, such as the number of internal registers or the amount of current drawn from each read or write access, to further adapt the architecture to the design specifications. In case this level of characterization is still not sufficient to properly represent a logic or power component, the modules' behaviour and energy model can be adapted to the requirements with the aid of some knowledge of the C++ and SystemC languages.

3.2.1 SystemC

SystemC [26] is a system-level modeling language designed to aid developers during the design and verification phases of the system architecture, independent of any detailed hardware and software implementations. This higher level of abstraction enables considerably faster, more productive trade-off analysis, design, and redesign than is possible at the more detailed RT level. It is built in standard C++ by extending the programming language with the use of class libraries. Through these extensions, it provides an event-driven simulation interface, where concurrent processes can be defined and interact with each other in a real-time environment.

3.2.2 SystemC-AMS

SystemC-AMS [27] is an extension to the base SystemC language that introduces system-level design and modeling capabilities for embedded Analog/Mixed-Signal (AMS) systems. Through this extension, it is granted the possibility to create mixed-signal virtual prototypes to model more analog-based applications, such as communication, automotive and sensor applications. Between the various features introduced, one of the most notables is the Timed Data Flow (TDF) computation model, which allows to describe continuous signals in a discrete-time manner.

3.2.3 Integration of GVSoC with SystemC-AMS

GVSoC is integrated within MESSY like any other component: it is encapsulated inside a SystemC module and instantiated at the start of the simulation, together with all the other system blocks. After instantiation, while SystemC manages the remaining blocks by executing them once, GVSoC initializes its event queue by starting software execution. The simulation then proceeds by alternating the execution of GVSoC and SystemC-AMS. In detail:

1. the functional model of the core invokes the *step_until()* API function of GVSoC, which executes the SoC functionality and returns the timestamp of the following event in the GVSoC queue;

2. the core executes a SystemC-AMS *wait()* until the next GVSoC event, to allow the execution of other SystemC-AMS components and the temporal alignment of the two simulators;

3. these two steps are repeated until the simulation is complete.

Whenever GVSoC needs to communicate with an external component, it does so by propagating the request's information to the functional model of the core through an *AXI_Request()*. The SystemC model then sends the data through the functional bus to the target block. The inverse process is applied when external components want to exchange information with GVSoC, with the only difference being the usage of *AXI_Response()* to initiate a communication with the ISS. A more detailed view of the system framework, together with the API functions used to interact with the functional simulator, can be seen in Figure 3.2.

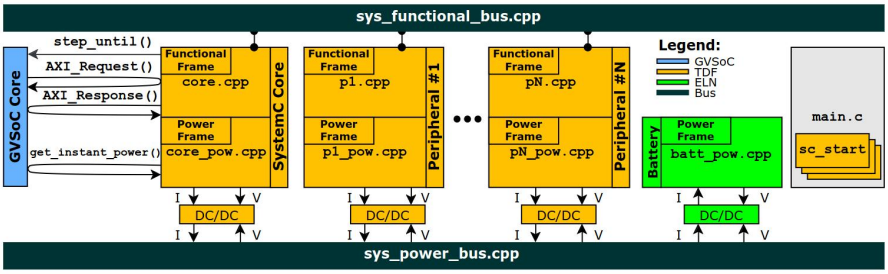


Figure 3.2: Framework integration between GVSoC and SystemC-AMS [1]

3.3 Webots: a robotic systems simulator

Webots [3] is an open-source professional robot simulation software. It provides a virtual prototyping environment that allows user to create 3D worlds with physic

properties such as mass and friction, which developers can use to design, program and test simple to complex robotic systems. Different types of robots can be created inside the environment, thanks to the wide number of objects and properties at user's disposal. Moreover, they can be equipped with several sensors and actuators, such as cameras and distance sensors, to better simulate their behaviour in working conditions. In case the software needs to be used only as a testing environment, it offers several robot models from the most important manufacturers. It also provides various interfaces to import custom models defined using specialized software (i.e. Blender, SolidWorks).

A Webots simulation is primarily composed of a world file and one or more controller programs. The former is a description of the properties of robots and their environments, managed using a hierarchical structure. The latter, instead, are computer programs written to control the robots placed in the virtual world. Each robot is assigned only one controller file. Controllers can be written using five different programming languages: C, C++, Java, Python and MATLAB. This gives developers more flexibililty over the code implementation, due to the different capabilities offered by each language and its libraries. For example, by using Python or MATLAB, it becomes much easier to implement a machine learning model or perform sensor data analysis. It is thanks to this amount of freedom that Webots was chosen as the robotic simulator for this thesis, since other popular software programs, such as ROS [28, 29], do not allow for custom programs to be executed by virtual prototypes.

3.4 Similar works

Although several implementations of the AXI bus protocol exist in hardware description languages like VHDL, Verilog and SystemVerilog [30, 31], only one, as of the time of writing, has been realized for the SystemC and SystemC-AMS languages. The SystemC-Components library [32], developed by MINRES Technologies, provides all the interfaces and transaction components to fully integrate an AXI-based bus into an existing system. The library implements the data exchange through the Transaction Level Modeling (TLM) 2.0 standard interfaces and traces all the involved signals, offering essential debugging capabilities for designers. Even though the library implements every aspect of the protocol, such as channel arbitration and out-of-order transactions, it was decided not to integrate it inside MESSY due to the high-level of abstraction it is based on. Since MESSY aims at simulating as accurately as possible the power consumption of the system's modules and its overall effect on the system's battery, it was deemed more appropriate to create a custom-made bus to manually manage the timing information. However, the library was kept as a reference during the bus implementation.

As for connecting GVSoC to Webots, the topic has been explored in only a single publication. GAPBOTS [33] is a Python program that creates a direct connection between the two simulators through the *multitasking* library. It does not launch the programs themselves, rather, it connects to them through their exposed APIs using two separate threads. The simulators are programmed to execute their tasks independently and exchange data only when GVSoC finishes processing its instructions, since it is the slowest between the two simulators. At that point, the most recent sensor acquisitions are written inside the functional simulator's main memory through the *mem_write()* API function, to decide the next actions of the robot, while Webots reads the current output from the memory through the *mem_read()* function. Even though the program works, it was decided not to base the thesis' implementation on such approach, given the inconsistency of the memory access times reported by the publication itself. In addition, since the thesis revolves around MESSY, it was decided to harness its capabilities and build the connection upon them, instead of creating an external program to control the flow of operations of the two applications, which would create an excessive layer of complexity. The implementation of the communication channel, reported later in more detail, was carried out in collaboration with another master thesis' student, whose work also revolved around MESSY.

Chapter 4

Methodology

This chapter focuses on the decisions, tools and techniques used to implement the communication channel between the two simulators and the system bus protocol. It is organized in two sections, each revolving around one of the main topics. Section 4.1 discusses the motivations behind the necessity of the link and the desired example use case. It then describes the realized architecture and details all the undertaken steps. Finally, it shows the achieved simulation flow. Section 4.2, instead, details the process of converting the high-level functional bus, connecting all of MESSY's modules, into a real, protocol-based, low-level functional bus.

4.1 Moving the Panda through MESSY

MESSY is a simulator that handles both the functional and extra-functional aspects of an embedded system running a custom application. However, due to its implementation, it operates entirely within an **isolated environment**: the energy consumption of each module is fixed, while sensor data are provided by designers, making systems easier to test and validate, but, at the same time, less realistic. In addition, the behaviour of more complex systems might be harder to evaluate with a command-line simulation environment. For example, a vacuum cleaner robot contains an embedded system responsible for managing its movement. In the case that such a system is the target of a MESSY simulation, when the machine encounters an obstacle on its path, it might be difficult to understand how the situation was handled and what decisions were made based solely on raw data. The goal of the first part of this thesis was to develop a method for MESSY to become a more **open** virtual prototyping framework. As a consequence of this, the simulator will be able to take part in a more detailed environment, allowing external applications to drive sensor data or receive the results computed by GVSoC for further analysis and usage. Thus, the simulated system will not only

comprise its computational part, but it will also integrate mechanical and electrical components, making the simulation more complex and realistic. Additionally, simulations can be made more accessible through data exchanges with graphical interfaces. To demonstrate the capabilities of the improved architecture, the framework was connected to Webots, tasked with managing the movements of a Franka Emika Panda in a pick and place scenario. The arm is equipped with a camera sensor, which produces images that a RISC-V processor analyzes using an object classification network to determine the location of a target block. Once the space coordinates have been determined, the robotic arm moves to its location, ready to pick up the block and transport it to another location. Several steps were taken to build this setup:

1. the virtual environment containing the Franka Emika Panda and the target block was created within Webots;

2. a library to create and use sockets to host the communication channel was built and integrated within MESSY's modules and the Webots arm's controller;

3. a kinematics library was implemented in the robot's controller, to automate the movements of the machine within the simulated environment once the target coordinates are received;

4. a camera and a controller peripherals were added to the system simulated by MESSY. Their implementation targeted the communication with Webots to exchange the necessary data for the pick and place application;

5. the controller of the robotic arm and the program run by GVSoC were created. The controller is programmed to wait for commands from MESSY before performing any kind of action, while GVSoC runs an application that requests data from the camera and controls, based on the object detection network results, the joints of the robot.

To attain the aforementioned flow, all components and structures built in MESSY are based on the C++ and SystemC constructs, while the controller for the Webots instance of the robotic arm was written in Python due to the library that was selected for managing kinematic operations, Robotics Toolbox for Python. The virtual environment in Webots, although very simple, was built from scratch. Due to a lack of trained object classification networks for the task at hand, and since the thesis did not focus on modeling and training one, GVSoC executes a *wait* statement that simulates the inference time of a potential network and delivers the exact coordinates of the object to pick to the robotic simulator. In the following subsections, each major step of the work is detailed: Subsection 4.1.1 briefly describes the virtual environment built within Webots to host the robotic arm and

the target block; Subsection 4.1.2 focuses on the connection between the simulators; Subsection 4.1.3 details the process of using inverse kinematic solvers to automate the movement of the arm; Subsection 4.1.4 presents the sensor modules added in MESSY, their behaviours and their communications with Webots; Subsection 4.1.5 details the general flow of the controller and GVSoC's program executions. It then shows the simulation flow for the completed pick and place application.

4.1.1 Creating the virtual environment

The virtual environment was built from scratch within Webots, hosting only the nodes required for the example application: the Franka Emika Panda robot model and the target object that the machine will reach during the simulation. Since the robotic arm is typically used for pick and place applications involving small objects, the scene nodes were placed on top of a table. The block dimensions were determined based on the maximum width of the Emika Panda's gripper. Given that the gripper measures $80mm$, the block is $50mm$ by $20mm$. Figure 4.1 shows the constructed virtual environment.

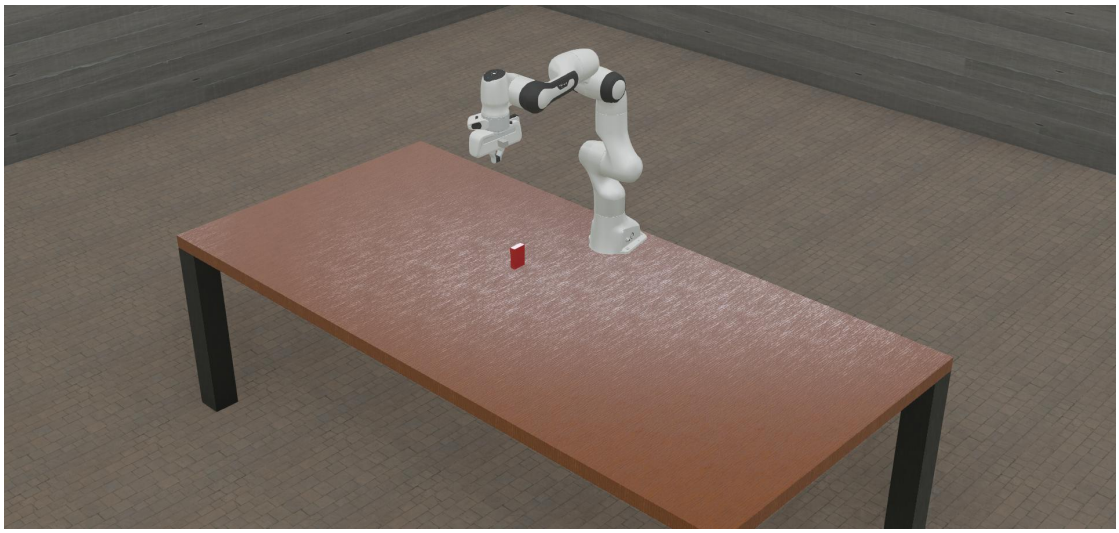


Figure 4.1: Virtual environment realized in Webots

A camera module was attached to the end-effector of the robot, the Franka Hand, to take pictures of the desk, which will then be analyzed by GVSoC to locate the position of the target block. It consists of a small block that contains a camera node. The module is represented in Figure 4.2, where the camera node is highlighted in orange. Whenever the cameras are mounted on the robot's end-effector (eye-in-hand camera) and there are no external ones to observe the workspace (eye-to-hand camera), depth cameras are used to assist the detection networks in determining

the space coordinates of the target object. Since Webots does not support depth cameras, and due to the lack of the object detection network within the simulation flow, a simple RGB camera, following the specification of an Intel RealSense D455[1], was used. The resolution of the produced images is 1280×800.

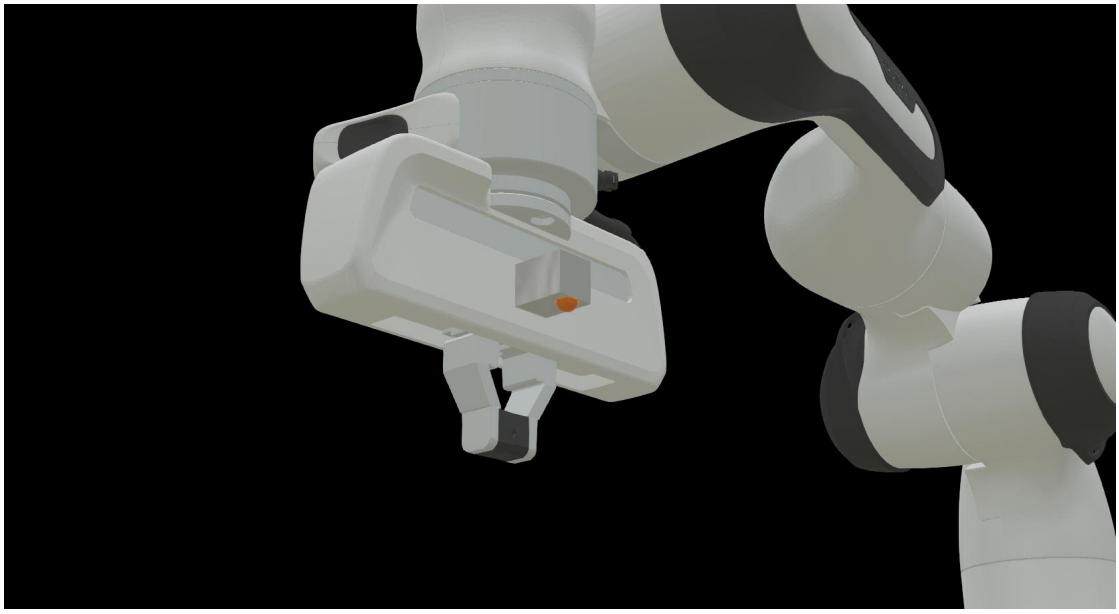


Figure 4.2: Camera module attached to the robotic arm

The timestep of the simulation was lowered from the default value of $32ms$ to $10ms$ to improve its accuracy, while the Emika Panda was programmed to move for twice that duration to create smooth motions.

4.1.2 The VirtualConnector library

To establish the connection between the simulators, sockets were employed. Since their structures are part of every operating system, calling their APIs does not require any external components or libraries and provides minimal performance overhead. Additionally, due to their extensive use in web applications, they are widely supported. To further reduce performance overhead, Unix sockets were chosen, at the cost of limiting MESSY to be connected only with locally available software. Two additional design choices were made to avoid excessive complexity in the connection's implementation: MESSY can be connected to one application only and, due to the framework requesting external data to better model sensors

[1] `https://www.intelrealsense.com/depth-camera-d455/`

and providing results for external usage, it was chosen to act as the client in the network architecture. To facilitate the use of the socket for communications, a small library, named VirtualConnector, was created. It contains two classes:

- **ConnectionConfig**, which hosts the file descriptors of the socket and handles its creation, initialization and termination;
- **VirtualConnector**, which manages the read and write operations that will occur through the socket.

At the start of the simulation, MESSY instantiates a ConnectionConfig object and, through the *initialize_connection()* method, it creates a socket and initializes its connection to the external application. Then, it assigns the instance to every module connected to the functional bus, except for the core, allowing each of them to exchange data with the other application. The core was ignored since the synchronization between the functional and extra-functional simulations could be impacted by requests from the external software. The implementation of the creation and initialization procedure of the socket follows the description mentioned in Subsection 2.4.1 and it is shown in Listing A.1.

During the simulation, any module can request data or send data through the channel, thanks to the *read_from_channel()* and *write_on_channel()* functions exposed by the VirtualConnector class. However, each transmission consists of two operations: a write transaction followed by a read operation. The former is used to indicate to the other program which information are required by the sensor or to deliver the data processed by GVSoC, while the latter is used to receive the requested information or an ACK message, to verify that the data were correctly received by the external software. Each read and write operation further consists of two steps:

1. in case of a write operation, the number of bytes of the message is sent as an integer value, while for a read operation, four bytes are read from the channel to determine the amount of bytes that compose the incoming message. This value allows for knowing in advance how much data to expect on the channel and for allocating the correct amount of memory to host the buffer;
2. the data are read or written on the channel.

To ensure that the connection can be established with any potential actor, the library sends data using the JSON format and expects packets with the same structure during read operations. Since C++ does not have a built-in library for creating and handling JSON objects, an external one was included. The Nlohmann JSON library [34] was chosen due to its simplicity and its handling of the *json* data type, which employs an approach similar to that of Python. Listing A.2 and Listing A.3 show how the two operations were implemented in MESSY.

To simplify the controller in Webots, a VirtualConnector library, encapsulating the aforementioned methods to handle the socket structure, was also written for Python. The implementation of the read and write methods can be found in Listing A.4 and Listing A.5.

4.1.3 Automating the trajectory of the Panda

Computing the position of the joints to reach specific coordinates in 3D space requires solving complex inverse kinematics problems, which involve differential equations and Jacobian matrices [35, 36]. For this reason, a library implementing solvers for this kind of problems, that supported the chosen robotic arm model, was integrated into the Webots controller. Robotics Toolbox for Python [22] was chosen for its simple and well-documented approach on handling different robotic arms. Furthermore, its solvers are based on the specifications and virtual models provided by manufacturers, making the obtained results more accurate. In particular, the information regarding the Emika Panda manipulator are based on the URDF file provided by the *franka_ros*[2] package. URDF is an XML format for representing a robot model, commonly used by Gazebo simulator and the Robotic Operating System (ROS) tools. Unfortunately, the version of Webots used during the thesis work (R2023b) presented an arm model different from the one included inside the Python library. The two of them, in fact, had different rotating axes for some of the joints, making it difficult to adopt the results computed by the library without further elaborations. To remedy this issue, the model used by the library was compiled from scratch and imported as a new Robot entity in Webots to replace the original one. Since the repository stored the robot model in a Xacro file format, an XML extension for describing how the joints of a robot are connected to create its structure, two compilation steps were required to produce the PROTO file needed to import the model in Webots. A PROTO file contains the same information about the structure of the robot as the Xacro file, but it specifies additional properties used by the virtual environment to improve the simulation of the robot's mechanical structure. First, ROS was used to compile the Xacro files into a URDF description. Then, using the *urdf2webots*[3] Python library, the URDF model was converted in a PROTO file that could be imported by Webots.

Once the manipulator was correctly imported into the virtual environment, it was time to integrate the library to manage its movements. The computation of the movements' trajectories and the relocation of the joints were implemented in a standalone function, called *move_arm()*, which could be invoked at any point

[2] `https://frankaemika.github.io/docs/control_parameters.html`

[3] `https://github.com/cyberbotics/urdf2webots/`

during the controller's execution to improve code flexibility. It requires only one argument: a list of three values detailing the target coordinates. The method is shown in Listing 4.1.

```
def move_arm(final_position):
    arm_pos = []
    for i in range(7):
        arm_pos.append(sensors[i].getValue())

    panda = rtb.models.DH.Panda()
    panda.q = np.array(arm_pos)

    # Compute transformation matrix
    x = final_position[0] - arm_base.getPosition()[0]
    y = final_position[1] - arm_base.getPosition()[1]
    z = final_position[2] - arm_base.getPosition()[2]
    T_matrix = SE3.Trans(x, y, z) * SE3.OA([0, 1, 0], [0, 0, -1])

    # Search for a valid solution to the inverse kinematic problem
    not_valid = True
    while not_valid == True:
        not_valid = False
        inversek_sol = (panda.ikine_LM(T_matrix)).q
        for i in range(7):
            if not (motors[i].getMinPosition()<inversek_sol[i] and
                    motors[i].getMaxPosition()>inversek_sol[i]):
                not_valid = True

    # Perform the movement until it is finished
    for i in range(7):
        motors[i].setPosition(inversek_sol[i])

    for i in range(7):
        curr_pos = motors[i].getPositionSensor().getValue()
        while (((abs(curr_pos-inversek_sol[i]))<=THRESHOLD)==False):
            arm_node.step(TIME_STEP)
            curr_pos = motors[i].getPositionSensor().getValue()

    return 0
```

Listing 4.1: *move_arm()* function implementation

Initially, the function retrieves the rotation angle of each joint and instantiates the virtual model of the Emika Panda provided by the library. Then, it assigns the recovered angles to the virtual model to establish the starting position of the movement. Next, it computes the transformation matrix, which combines information about the translation and rotation movements to be performed to reach the end point. To avoid unreachable position errors when the goal is within

range, the coordinates of the end point have to be relative with respect to the base of the robot. After providing the model with the starting position of the robot and the desired transformation, the solution of the inverse kinematics problem is computed through the *ikine_LM()* method of the virtual model. Before using the received output, a verification step is performed to make sure that every joint can perform that specific motion. Therefore, each new angle is compared against the minimum and maximum rotational limits of the interested joint. Whenever a non-valid solution is found, the method is called again, to recover another possible set of values to be tested and, possibly, used. Finally, the Webots manipulator is moved according to the resulting values.

4.1.4 Implementing the peripherals in MESSY

In order to build the proposed scenario, two peripherals were added to the embedded system simulated by MESSY:

- a **camera** sensor, to handle the information provided by the module placed on the end-effector of the robot;
- a **controller** module, to simulate the dialogue between the main processor, which runs the network inference, and a secondary microcontroller, which solves the inverse kinematics problem and controls the motors.

These peripherals were integrated into the system through their addition to the list of system components, specified in a configuration file using the JSON data format. For each generated module, a header file and a source file were created. The former hosts the description of the hardware block connected to the system's bus. It contains the list of the input, output and internal signals, the list of processes to be simulated, as well as the variables and class methods. The latter, instead, is used to implement all those introduced methods. The files created during the compilation of the framework are provided with all the necessary functions to properly connect to the functional and power buses, together with the management of read and write operations on internal registers. After recompiling MESSY to generate and integrate these peripherals, their behaviour was customized to include the necessary operations for the pick and place application.

The camera peripheral makes use of a control register to decide which action to perform. In this specific application, the module is responsible only with retrieving the image data from Webots, therefore a single control bit sequence was defined. Whenever a write operation occurs on the control register of the module, its bits are checked and, if the new value matches the bit sequence for the "new image" command, an image request is sent to Webots using the *get_camera_image()* function. This method harnesses the VirtualConnector library's capabilities to

communicate with the robotic simulator. First, it creates the JSON structure containing the request and sends it. Next, it reads the channel to store the image information. Finally, it unpacks the JSON structure and stores its data in the internal registers using a dedicated function called *json_parser()*. The two methods are displayed in Listing 4.2 and Listing 4.3.

```
void Sensor_camera_functional::get_camera_image() {
    json request, data;

    // Send request
    request["command"] = "GET_IMAGE";
    VirtualConnector::write_on_channel(socket_fd, request);

    // Await data
    data = VirtualConnector::read_from_channel(socket_fd);
    json_parser(data);
}
```

Listing 4.2: *get_camera_image()* function implementation

```
void Sensor_camera_functional::json_parser(json in) {
    int width, height, index;
    std::vector<unsigned char> image, image_default;

    // Loop through the list of objects
    for (auto& item : in.items()) {
        if (item.key() == "camera") {
            width = item.value().value("width", 0);
            height = item.value().value("height", 0);

            image_default.clear();
            image = item.value().value("image", image_default);

            /* The pixels information arrive as four chars:
             * R (0) | G (1) | B (2) | Alpha (3)
             */
            for (int i = 0; i < (4 * width * height); i++)
                register_memory[i + DATA_REG_OFFSET] = image[i];
        }
    }
}
```

Listing 4.3: *json_parser()* function implementation

After completing the transmission and handling the received data, a status register is updated to signal to GVSoC that the data can be read and used for the inference task.

The controller peripheral follows the same implementation structure as the

camera module, but it manages two operations: sending the target coordinates to the robotic manipulator and commanding Webots to stop the simulation. The first operation is executed by the *set_ee_position()* function, while the second one is requested by the *set_end_program()* function. Both methods follow the same procedure: first, they create the JSON data structure with the request and send it to Webots; then, they wait until the simulator produces an ACK response and sends it back. The ACK message transports an integer value indicating the status of the request. For the movement of the end-effector, two statuses can be expected: the movement could not be completed, indicated by a value of 0, or the movement was completed successfully, indicated by a value of 1. The conclusion of the robotic simulation, instead, can generate only a single response, indicated by the value of 0: simulation terminated. The received status values are converted by the two functions to avoid reporting a 0 to the core, interpreted by GVSoC as if no update has been received.

```
int Sensor_controller_wrapper_functional::set_ee_position() {
    int ret = -1;
    json request, pos, data_rec;
    controller_t *wrapper = (controller_t*) register_memory;

    // Send request
    pos["x"] = wrapper->x;
    pos["y"] = wrapper->y;
    pos["z"] = wrapper->z;
    request["command"] = "MOVE_EE";
    request["position"] = pos;
    request["time"] = wrapper->time;
    VirtualConnector::write_on_channel(socket_fd, request);

    // Await ACK
    data_rec = VirtualConnector::read_from_channel(socket_fd);
    for (auto& item : data_rec.items()) {
        if (item.key() == "mov_end") {
            if (item.value() == 0)
                ret = 1;
            else
                ret = 2;
        }
    }
    return ret;
}
```

Listing 4.4: *set_ee_position() function implementation*

After each operation is completed, the status register is updated to indicate to GVSoC that the data were correctly received by Webots. Listing 4.4 and Listing 4.5 show the implementations of the two functions.

```
int Sensor_controller_wrapper_functional::set_end_program() {
    int ret = -1;
    json request, data_rec;

    // Send request
    request["command"] = "END_PROGRAM";
    VirtualConnector::write_on_channel(socket_fd, request);

    // Await ACK
    data_rec = VirtualConnector::read_from_channel(socket_fd);
    for (auto& item : data_rec.items()) {
        if (item.key() == "ack") {
            if (item.value() == 0)
                ret = 1;

            break;
        }
    }

    return ret;
}
```

Listing 4.5: *set_end_program() function implementation*

4.1.5 Simulation flow

After implementing all of the modules and structures needed for the communication and simulation processes, the programs launched by GVSoC and Webots to manage the flow of the pick and place example application were implemented. The robot manipulator's controller is initially tasked with creating the server socket, listening for an incoming connection from MESSY and initializing all the sensors and actuators of the robotic arm. Afterwards, it enters an infinite loop, waiting for any command from the main controller. When a command is received, it is parsed and executed. The loop is exited only after receiving the "END_PROGRAM" request from MESSY. Once the loop is terminated, the channel is closed and the simulation concluded. The infinite loop is shown in Listing 4.6. The GVSoC program, instead, starts by declaring pointers to the memory addresses of control, status and data registers of the two peripherals. Once the relevant memory regions can be accessed, the image of the camera module is requested from Webots. After storing the pixels' information in the internal registers of the camera module, the inference of the object classification network is simulated through a *pi_time_wait_us* statement. Then, the position of the target block is stored within the registers of the controller module and the command to move the end-effector is sent to the robotic manipulator. Once the movement is concluded, the simulation is stopped and the program terminates.

The complete code implementation of the executed program can be found in Listing 4.7.

```
while robot.step(time_step) != -1:
    data_rec = None
    data_send = dict()

    # Wait for a request from MESSY
    data_rec = virt_connector.receive_message()

    if data_rec["command"]:
        # Based on the command received, send back different data
        if data_rec["command"] == "GET_IMAGE":
            data_send["camera"] = self.json_camera()

        elif data_rec["command"] == "MOVE_EE":
            pos = [data_rec["position"]["x"],
                   data_rec["position"]["y"],
                   data_rec["position"]["z"]]
            data_send["mov_end"] = self.move_arm(pos)

        elif data_rec["command"] == "END_PROGRAM":
            data_send["ack"] = 0
            stop_execution_request = True

    # Send the requested data to MESSY
    virt_connector.send_message(data_send)
    if stop_execution_request == True:
        break
```

Listing 4.6: Emika Panda's controller main loop

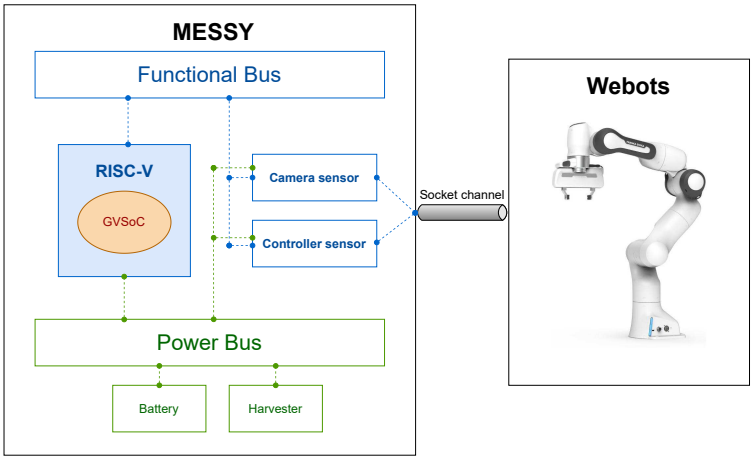


Figure 4.3: System architecture for the pick and place scenario

```c
int main(void) {
    uint8_t* camera_ctrl = (volatile uint8_t*) (CAMERA_CTRL_OFF);
    uint8_t* camera_status = (volatile uint8_t*) (CAMERA_STATUS_OFF);
    uint8_t* camera_sensor = (volatile uint8_t*) (CAMERA_DATA_OFF);

    controller_t wrapper_data;
    controller_t* controller_data = CONTR_OFF;
    uint8_t* controller_ctrl = (volatile uint8_t *) (CONTR_CTRL_OFF);
    uint8_t* controller_status = (volatile uint8_t *) (CONTR_STATUS);

    // Request image and wait until data is recovered
    *camera_ctrl = 0x01;
    while((*camera_status) != 0x01);
    *camera_ctrl = 0x00;

    pi_time_wait_us(INFERENCE_TIME);

    // Write red block position
    wrapper_data.control = 0x00;
    wrapper_data.status = (*controller_status);
    wrapper_data.x = X_POS;
    wrapper_data.y = Y_POS;
    wrapper_data.z =  Z_POS;
    *controller_data = wrapper_data;

    // Send the data
    *controller_ctrl = 0x01;
    while((*controller_status) == 0x00);

    // Close Webots connection
    *controller_ctrl = 0xFF;
    while((*controller_status) != 0x01);

    return 1;
}
```

Listing 4.7: Program executed by GVSoC

After implementing the two programs, the scenario can finally be simulated. To properly launch the simulation, Webots must first start the manipulator's controller to create the server socket. Then, the functional and extra-functional simulations managed by MESSY can be launched, establishing a communication channel by connecting to the socket. The overall system architecture can be seen in Figure 4.3, with the functional and power aspects are highlighted using different colors. Figure 4.4, on the other hand, graphically represents how communication between the two software systems occurs and the sequential order of the operations involved. Whenever one of the operations demands only a specific framework to be completed,

the other application is put on hold.

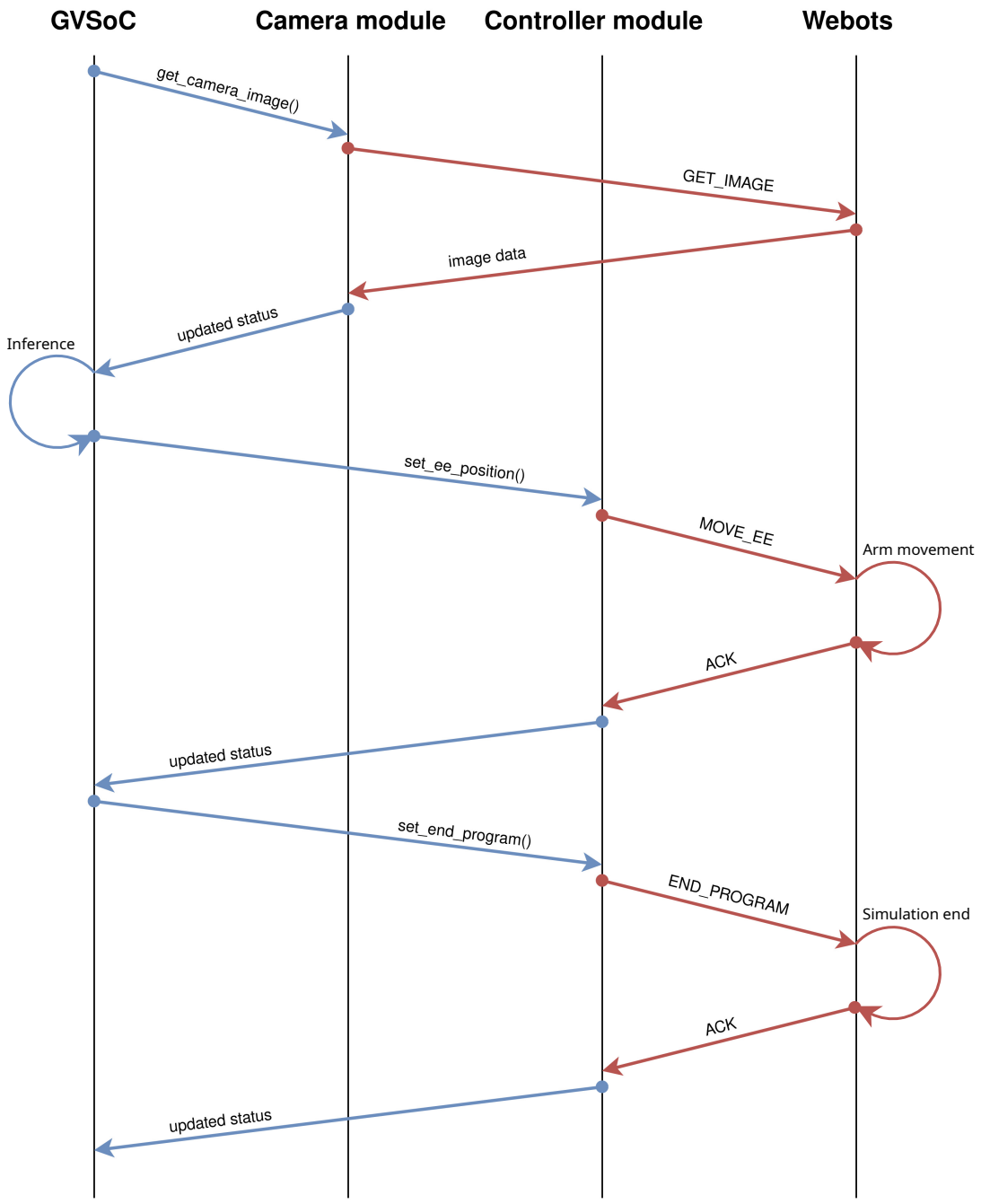


Figure 4.4: Operations' flow between the two simulators

4.2 Improving the functional bus

Although MESSY tries to simulate an embedded system as accurately as possible, the results produced by the framework differ from the ones measured on a physical system due to its high-level implementation. The execution times of simulated tasks might vary because of the fixed timings of read and write operations on internal registers or the instantaneous communications occurring between modules, a consequence of the ideal implementation of the functional bus. The power consumption of the peripherals does not transition realistically between different states but changes abruptly between fixed values, indicated through the JSON configuration file. Therefore, even if the characteristics and behaviour of the battery are properly simulated, its capacity over time and the current drawn by the system's modules are inaccurate, causing the system to operate longer or shorter than expected. Thus, the second half of this thesis focused on improving the framework to produce more realistic results. In particular, the communication mechanism of the system was entirely reworked to establish a real, protocol-based, low-level functional bus. The new interconnection was based on one of the most used protocols in the embedded domain: AXI. To build and integrate the low-level bus into MESSY, the procedure described below was followed:

- the manager and subordinate interfaces were implemented separately;
- a small system, composed of two entities, was built to test and validate the interfaces and the transmissions occurring through them;
- the test system was extended through the addition of a bus module, and all the communications between the two entities were forwarded through it;
- the functional bus and the interfaces were integrated into MESSY.

In the following subsections, the aforementioned steps are detailed: Subsection 4.2.1 describes the architecture of a system characterized by point-to-point AXI communications and focuses on the implementation of the master and slave modules, as well as the underlying software structures; Subsection 4.2.2 extends the base implementation with the addition of a bus module; Subsection 4.2.3 details the process of integrating the communication mechanism into the simulation framework of MESSY.

4.2.1 Building a point-to-point communication system

The communication system and its software structures were implemented so that they could be configured and integrated into the simulated system through the same process as the other modules. To further facilitate the system's generation, it was

decided to separate, as much as possible, the management of the communication signals and processes of each module from the internal signals, variables and procedures used to describe its behaviour. As a consequence, each block has the structure shown in Figure 4.5: it is described by a main construct, which incorporates the ports of the manager and subordinate interfaces, as well as the module that details the block's behaviour.

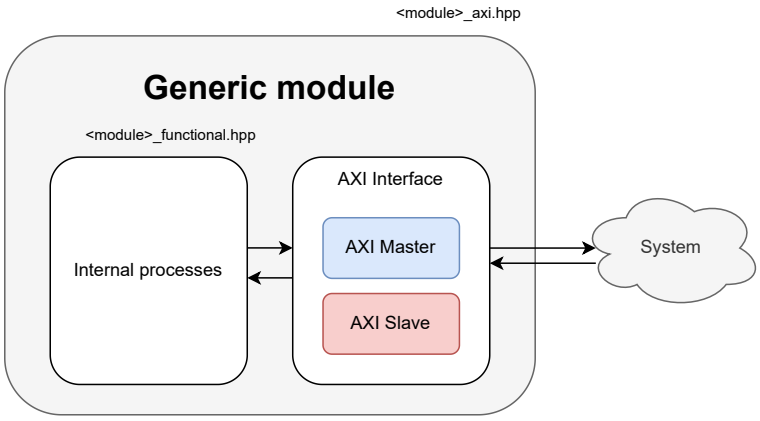


Figure 4.5: Module implementation

A header file was created to host common signal and interface definitions for the master and slave blocks. In this file, two structures containing the definition of the input and output signals of the five channels employed by the protocol were defined. The structure used by the manager module is represented in Listing A.6, while the one employed by the subordinate block can be easily obtained by inverting the manager structure's signal types. Additionally, two virtual classes were created to define the methods that have to be called or implemented by the block's behavioural description to properly exchange information with other modules: *Master_2_AXI_Port* and *AXI_2_Slave_Port*. Depending on the role of the module within the architecture, one or both interfaces must be inherited. The former class targets hardware blocks that make read or write requests to other system modules, such as the core of the architecture or a DMA controller, and defines two callable functions:

- a **read** function, to request a specific amount of bytes from a given address;
- a **write** function, to send information at a specific address.

Both methods' inputs and outputs must be handled by the caller. Conversely, the latter interface targets modules that need to respond to those read and write requests. It contains functions that must be implemented by the inheriting class, such as:

- a **read** and **write** function, used by the subordinate process to request a load or store operation on internal registers;
- an **is_ready** function, used to check whether the communication request can be accepted or not by the peripheral;
- a **start_transaction** and **close_transaction**, used to block the execution of internal processes until the transmission is concluded, thus avoiding concurrent accesses or modifications to the registers' values.

The two interfaces were built in separate files as SystemC modules. The manager module inherits the *Master_2_AXI_Port* virtual class and implements its read and write operations as detailed by the AXI protocol (see Subsection 2.5.1 for more information about these two operations). Since the system simulated by MESSY, except for the functional ISS contained within the core, does not operate based on clock signals, while the protocol requires one due to having a low-level implementation, it was decided that the master interface should generate a clock signal and send it to the subordinate interface during the data exchange. The slave module, instead, is composed of only one method, which waits indefinitely for a read or write request. When one of these two requests is received, it awaits the conclusion of all internal operations, blocks them using the *start_transaction* method and, then, proceeds to handle the related channel's signals as described by the protocol. Finally, it restarts the internal processes.

As previously reported in Section 2.5.1, the AXI protocol was not fully implemented: only the base versions of the read and write operations were realized, with the list of all the included signals shown in Table 2.1. The protocol was tailored to the current state of MESSY. Since the framework supports solely a single core and, at present, there is no integrated DMA controller module, the architecture hosts only a single manager block. Thus, the arbitration mechanism, atomic access management and protection level support are not required by the system and would not only increase the complexity of the protocol implementation, but also make the simulation heavier and slower. Additionally, transactions were not implemented because read and write operations are executed sequentially, without interruption. Therefore, the protocol was integrated in a form suitable for the architecture, avoiding the addition of structures and procedures that are not used and would hinder its internal processes.

To test the two modules, a system composed of two blocks, a core and a generic sensor, was built. The core was tasked solely with making read and write requests to the sensor, therefore it was equipped only with a manager interface. The data and addresses provided to each operation were manually defined to facilitate the validation of the communication mechanism. The sensor, instead, only handled the received requests, thus requiring solely a subordinate interface. A simple process

simulating an internal workload through a *wait* statement was defined to test the correct execution of the *start_transaction* and *close_transaction* methods. The two blocks were connected directly using a testbench module, which was also tasked with monitoring the channels' signals. The system can be seen in Figure 4.6.

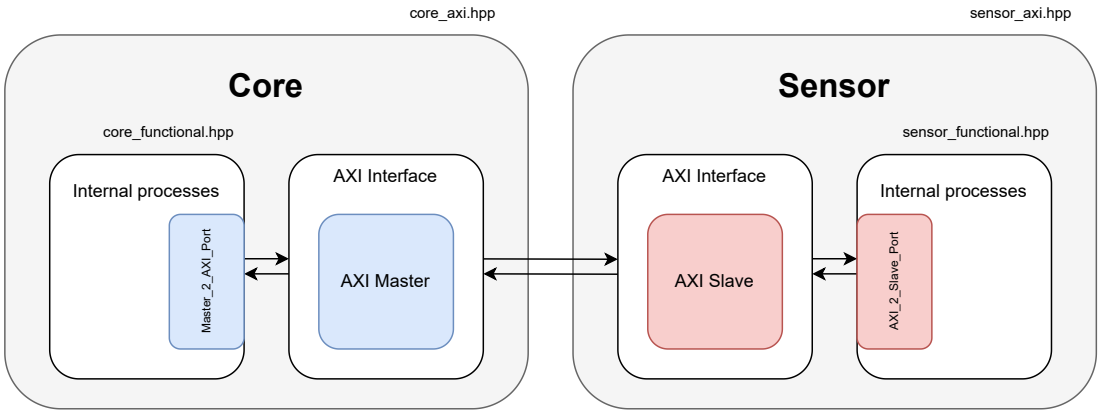


Figure 4.6: Point-to-point system architecture

4.2.2 Introducing a bus between modules

Since embedded systems do not allow different modules to communicate directly, but rather through a common interconnection, a bus was introduced to the aforementioned point-to-point architecture. Therefore, each transmission occurring between the core and sensor blocks had to pass through this channel. The updated system is shown in Figure 4.7.

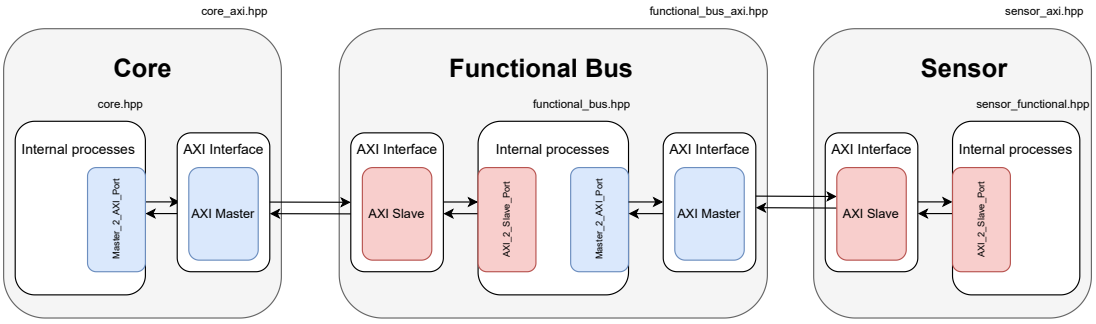


Figure 4.7: Bus-centric system architecture

The bus line was implemented with a structure similar to the one represented in Figure 4.5. It employs both manager and subordinate interfaces to properly handle read and write operations across the system. Whenever a module acting

as a manager initiates a communication, it sends the request's information to the functional bus through its slave interface. The interconnection examines the target address and activates the corresponding system component. It then acts as the manager, requesting or writing data to that component while storing the requested information and related responses. Finally, the bus reports the obtained data back to the original transmitter. A graphical representation of the described flow can be seen in Figure 4.8. The behavioural processes implement the routing capabilities of the interconnection.

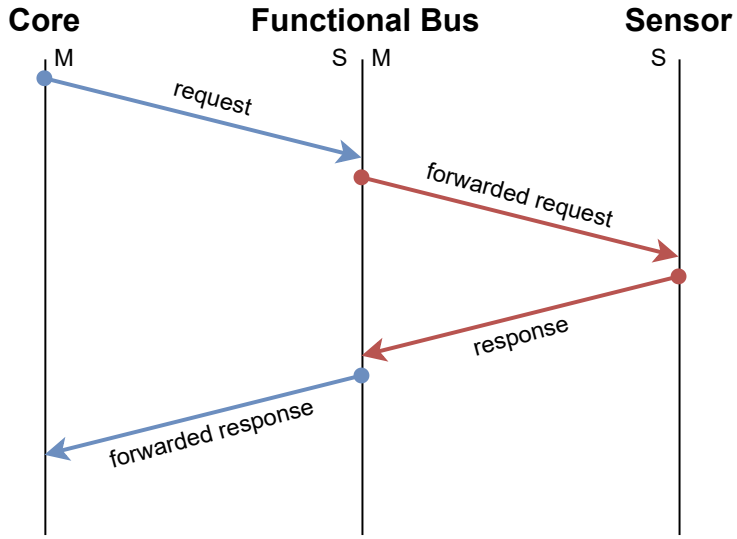


Figure 4.8: Flow of communication within the bus-centric architecture

To properly handle the communication process described above, a small memory buffer was introduced within the bus structure to store the data that the master wants to read from or write to at the destination addresses. It does not have a specific dimension, but, instead, it is allocated dynamically based on the amount of bytes that are involved during the transmission. The subordinate interface was also modified so that, after collecting the request's information, it signals to the internal processes that the transmission can be forwarded to the target unit, while the transmitter is put on hold. An additional virtual class defining the methods used to handle the routing operation and check its termination was defined: *AXI_2_Bus_Port*. It declares three operations:

- **is_response_ready**, to verify the status of the transaction between the interconnection and the destination module;
- **send_request**, to signal that a routing operation has to start;

- **write**, to store the data to be sent to the destination within an internal memory buffer.

With the introduction of the bus channel into the system architecture, the response signals of the Read Data (R) and Write Response (B) channels were finally assigned appropriate values. It was not done earlier because errors have to be managed by either the bus or the receivers, depending on the specific circumstances. The list of values that each response can assume is reported in Table 4.1. Since

Error code	Description
00	Normal access success or exclusive access failure
01	Only a portion of an exclusive access has been successful
10	Successful access, but the subordinate encounters an error
11	No subordinate block was found at the requested address

Table 4.1: Error codes for the RRESP and BRESP signals

exclusive accesses were not managed during the implementation of the protocol, the code error "01" cannot be received by manager modules. Instead, the "10" bit sequence is produced by subordinate blocks whenever the requested address cannot be accessed, while the "11" error sequence is generated by the functional bus whenever there is no peripheral associated with the received address. As an example, imagine to have a system with two sensors and the memory map shown in Table 4.2. In the case where the architecture's core wants to read five bytes

Module	Base address	End address	Memory register size
Sensor1	0x20000000	0x200000FF	256
Sensor2	0x20000100	0x2000010F	16

Table 4.2: Memory map of the system

starting at address 0x2000010E, the functional bus correctly identifies Sensor2 as the destination component, enables its communication module and initiates a read transaction. It requests five bytes, but only two of them can be accessed. As a consequence, the responses produced by the subordinate will be: 00, 00, 10, 10, 10. Instead, in the case where the read address is 0x20000115, the functional bus does not locate any subordinate blocks and responds to the transmitter with: 11, 11, 11, 11, 11.

To test the new bus-centric structure, a testbench instantiating two sensors and a core was created. The core is tasked with executing the same read and write operations as before across the two peripherals, while the two sensors simply

simulate a workload through a *wait* statement. All of the communication signals of these modules were connected directly to the functional bus's AXI signals, except for the subordinate interfaces' output signals. Since multiple sensors are attached to the interconnection, these signals are multiplexed based on the peripheral enabled by the functional bus's behavioural process.

4.2.3 Integrating the communication protocol into MESSY

Integrating the interconnection and the two interfaces into MESSY required several modifications to the original framework's structure. First of all, the original version of the functional bus was replaced by the newly created one, while the header files containing the software structures, along with the manager and subordinate interfaces used by the protocol, were simply added to the system. Next, the core module was rewritten to separate the communication-related signals and processes from those managing the functional ISS. The final structure of the processor block is similar to the one realized for the isolated testing environments, which is shown in Figure 4.6 and Figure 4.7. The updated functional model inherits from the *Master_2_AXI_Port* virtual class and employs its methods within the *handle_request* function to transmit data over the interconnection when requested by GVSoC. Then, the focus shifted to the peripherals. They were heavily reworked to achieve two goals:

- to enable the system's generation process of MESSY to create specific source files that host custom code for each hardware block, allowing procedures such as connecting to external applications or reading and writing files to be implemented without having to modify the communication and register file processes;
- to separate, as was done for the core module, the communication-related signals and processes from the behavioural ones.

To accomplish the first objective, a SystemC base class named *Sensor* was created. It contains the enable and power signals, as well as the register file memory pointer, a ConnectionConfig reference and delay variables. It primarily implements three methods:

- a function to implement the behavioural process of the peripheral, called **sensor_logic**. It contains an infinite loop where it continuously checks for any communication requests and, after resolving them, executes a **custom_logic** function, located within the custom source files generated by MESSY;
- a function to simulate the delays and energy consumption associated with the read and write operations of the peripheral, named **power_logic**. The

statements executed by this function are similar to the original portions of code contained within each sensor module, with the difference of using specific variables to store the configured consumption and delay values. The procedure must be called by the methods that manage the read and write operations on internal registers;

- a function to store the socket connection information, identical to the one originally contained within each sensor, named **set_connection_parameters**.

The peripherals generated by MESSY must inherit from this base class and define only the implementation of the custom routines, along with the values for the number of memory registers and delays for access operations, thereby separating the simulation aspects from the more abstract ones as much as possible. To separate the communication-related structures from the hardware block description, instead, a framework similar to that represented in Figure 4.6 or Figure 4.7 has been used. However, in order to maintain the source files provided to the designers without any non-custom aspects, the virtual *AXI_2_Slave_Port* class is inherited and implemented by the *Sensor* class. Within the interface implementation, power and timing information are considered during read and write register operations by calling the *power_logic* function. Since read operations are managed by subordinate blocks, the power procedure is called during each operation, whereas for write operations, it is called once at the end of the data stream for as many times as the bytes received. This approach ensures that the write delays do not hinder the execution of the communication protocol. A method, called *postwrite_move_simulation()*, was created and added to the virtual class for the subordinate interface to launch the power management functions after the reception of write data. Figure 4.9 shows a class diagram representing the relations between the involved classes.

After revising the peripherals' structures, the testbench was altered to instantiate the correct modules and to incorporate the multiplexer process for the interconnection's AXI manager input signals. The pick and place application built to test the connection between GVSoC and Webots described in the previous section was employed to validate the improved system architecture.

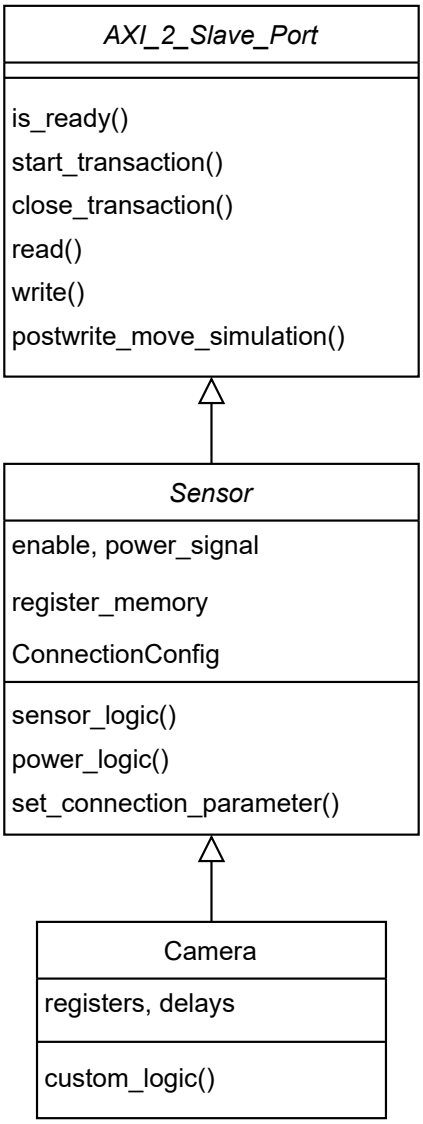


Figure 4.9: Camera module class diagram

Chapter 5

Experimental results

This chapter presents the experiments performed upon the modified architecture and the corresponding results. Several sections are used to separate the different tests: Section 5.1 details the configuration parameters used by the two software systems and describes the motivations behind them; Section 5.2 presents the measured execution times for the pick and place application with different configuration setups; Section 5.3 analyzes the case where the camera needs to be regularly accessed during the arm's motions for possible coordinates adjustments; Section 5.4 discusses the performance of the newly integrated interconnection compared to the original one.

5.1 Testing setup for the joint simulation

Apart from some modifications to the general flow of the simulated example scenario, which are discussed in the relevant sections, the same parameters were used to configure the Webots and MESSY environments for all of the tests. As already discussed in Subsection 4.1.1, the timestep of the robotic simulation, which defines the duration of a single simulation step in milliseconds, was reduced from the default value of $32ms$ to $10ms$ to improve the response times of the Emika Panda manipulator's controller and the accuracy of the machine's movements. The manipulator, instead, was programmed to have a movement timestep of twice that duration, allowing for smoother motions. Moving to the camera module that was attached to the end-effector of the robotic arm, it was configured based on the specifications of an Intel RealSense D455. Thus, the resolution of the generated images is 1280×800, while its field of view is $90 \times 65°$. Focusing on MESSY, the functional and extra-functional simulations were configured to terminate whenever they exceeded 1000 seconds, while the clock period for the AXI-based interconnection was set to $2ms$. Aside from the simulation aspect, the

configurations mainly revolved around the camera and controller peripherals. These sensors were added to the system and later configured through the same JSON file used to define the system's components. The JSON objects describing the two modules are shown in Listing 5.1 and Listing 5.2.

```
"camera": {
    "register_memory" : 8192000,
    "states":{
        "read" : {
            "current" : "0.12",
            "delay" : "30"
        },
        "write": {
            "current" : "0.16",
            "delay" : "30"
        },
        "idle":{
            "current" : "0.002"
        }
    }
}
```

Listing 5.1: Camera peripheral configuration

The camera module contains 8192000 byte registers, sufficient to hold control and status values, as well as to store the data for a single image. Each register takes $30ms$ to be accessed, regardless of the type of operation. The controller peripheral makes use of the same delay information, while it has a smaller register memory, comprised of only 256 byte registers.

```
"controller_wrapper": {
    "register_memory" : 256,
    "states":{
        "read" : {
            "current" : "0.12",
            "delay" : "30"
        },
        "write": {
            "current" : "0.16",
            "delay" : "30"
        },
        "idle":{
            "current" : "0.002"
        }
    }
}
```

Listing 5.2: Controller peripheral configuration

A final configuration was necessary to simulate the execution of an object detection network within the simulation flow. As mentioned in Section 4.1, due to a lack of trained neural networks for the pick and place application, GVSoC executes a wait statement that simulates the inference time of a potential network, after which it sends the coordinates of the target object to the manipulator. A wait time of $130ms$ was used, based on the inference latency of a Squeezed Edge YOLO [37] when simulated on a GAP8 processor through GVSoC.

5.2 Measuring the simulation times

As a starting point for evaluating the framework, the simulation times under various setups were measured. First, the time required to move the arm from its idle position to the target object was measured in Webots, without involving any communication with MESSY. Next, the two simulators were connected to measure the time needed to run an oracle version of the example application. In this scenario, MESSY already knows the location of the target block and transmits it directly to Webots, without requesting an image to infer it from. This setup is used to evaluate the additional time required by the overall simulation, compared to the Webots-only scenario, when a small amount of data is transferred between the two software systems. To deepen the analysis, measurements were taken on

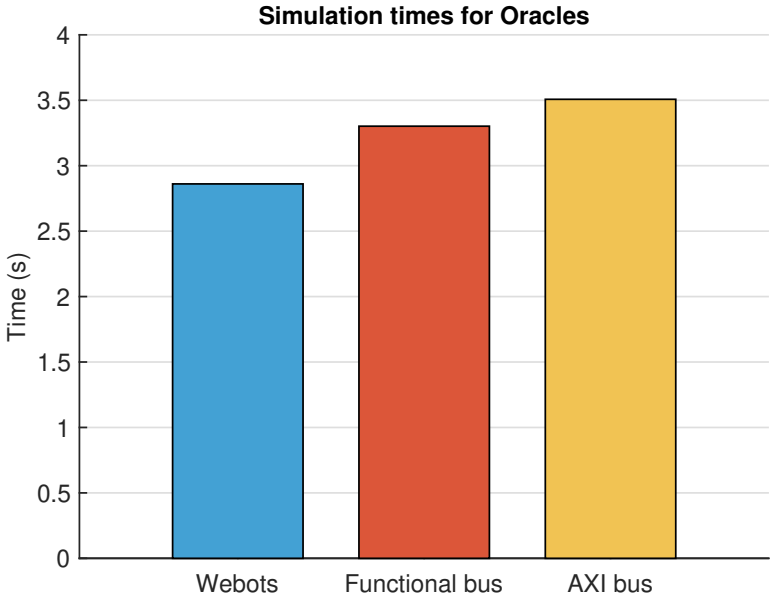


Figure 5.1: Oracles' simulation times

both the original implementation of the interconnection and its reworked version. The results, shown in Figure 5.1, demonstrate that the major contribution to the overall simulation is given by the computation of the trajectories and the movement of the robotic manipulator, while the management of the integrated system and the communication between the two simulators, occurring via socket, amount to an increase of simulation time between 10.91%, whenever the original implementation of the bus is involved, and 18.14%, in case the protocol-based interconnection is used. The usage of the AXI-based channel alone, without altering the previously described flow, increased the simulation time by about 8.13%, highlighting the impact of low-level signal exchanges. Whenever large amounts of data need to be transferred, as will be the case later, the presence of these exchanges increase will further separate the two simulations, becoming the leading contributor to the overall simulation time.

After this first test, the complete simulation, as originally planned (see Subsection 4.1.5), was launched and analyzed. As with the previous test, measurements were taken for the framework using both the original implementation of the interconnection and its reworked version. For this evaluation, however, the simulations were measured twice: first with the previously detailed flow, and then with the wait statements, used to simulate the presence of an object detection network within the simulation flow, replaced by the introduction of FLOPS instructions to simulate the operations that would have been executed by a real network. These operations do

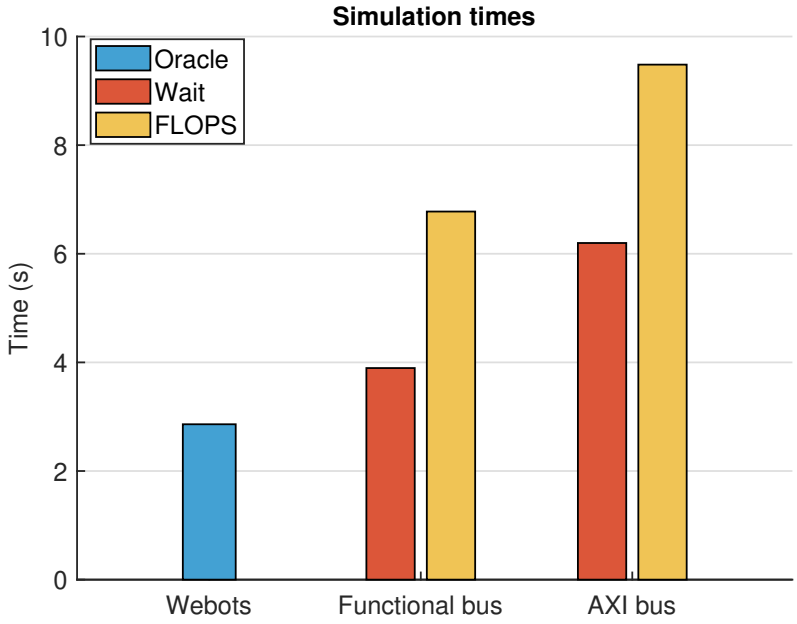


Figure 5.2: Simulation times for the complete pick and place flow

not involve any data from the camera module and are executed for the amount of time needed to advance the simulation of approximately $130ms$. The results of this test are represented in Figure 5.2. From this graph, it can be seen that the addition of an image request to the operations' flow was not as impactful as expected when the original channel was involved, even though a considerable amount of data (~ 18.5MB) was exchanged through the socket connection. The simulation time, in fact, increased of about 17.85% compared to the previous test. However, this value rose to 43.62% for the protocol-based interconnection, highlighting once again the significant importance of the low-level exchanges. An additional, yet crucial, observation can be made regarding the time differences between simulations using wait statements and those substituting them with FLOPS computations: the wait statement only advances the simulation's internal signals by the provided amount of time but does not perform any actual workload to simulate it. In fact, whenever FLOPS are integrated into the simulation, even though they advance it by the same time interval, the total execution time increases by approximately 3 seconds. In particular, the original framework implementation gains 2.88 seconds (42.52% of increase), while the AXI-based simulation gains 3.28 seconds (34.63% of increase).

5.3 Precision adjustments overhead

Most of the times, a single image is enough for an object detection network to accurately locate the position of a fixed target object within the environment. However, in some situations, the pick and place application may involve moving objects. In these scenarios, images need to be captured during the movement of the robotic manipulator in order to correctly track the shifting position of the target and adjust the machine's trajectory accordingly. Such possibilities were tested using the modified virtual prototyping framework by altering the operations' flow. For this test, the program launched by GVSoC executes an internal loop where an image is requested to Webots, parsed and the results are sent to the robotic arm, along with a new parameter indicating the time interval to dedicate to the arm's motions, before responding to MESSY, allowing to regulate the number of images to capture within a second. The loop is exited whenever the arm reaches the target coordinates. The test was performed on the framework using both the

	1 fps	**2 fps**	**4 fps**
Functional bus	3.6	5.6	12.4
AXI bus	3.8	6.5	12.6

Table 5.1: Average number of captures to terminate the arm's motion

original implementation of the interconnection and its reworked version and, since

a wait time of $130ms$ is used to simulate the object detection network, it measured the simulation times for one, two and four images captured per second on the two architectures.

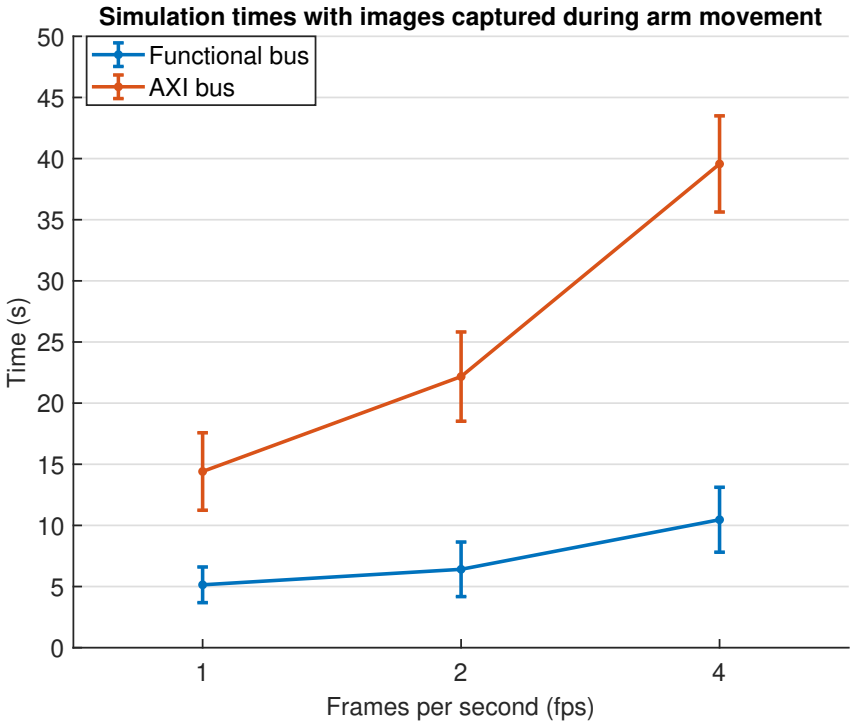


Figure 5.3: Measured times with varying number of captures

The results, represented in Table 5.1 and Figure 5.3, are computed as average values after running ten simulations on both architectures. Figure 5.3 also includes the standard deviation for each measurement, highlighting the amount of variation of the values from the computed mean. As it can be seen, the simulation with the functional bus that requests one image per second has a one-second overhead compared to the normal simulation scenario, the measurement of which is shown in Figure 5.2, while higher frames per second further increase this value, causing the simulation to run for more than twice the normal execution time. This increment can be explained by the additional capture requests made to Webots, which rose from an average of 3.6 to an average of 12.4. The same increment in number of captures, though, did not have the same effect on the simulation times for the AXI-based software system. Due to the low-level signaling occurring between the system's different units during communication, each new image request exponentially increases the overall simulation time. Specifically, when MESSY requests a single image per second, the simulation time is 2.3x the normal execution

time, while this value grows to 6.4x for four requests per second. The time differences of running this example scenario on the two framework implementations vary from 64.32% to 73.55%.

5.4 AXI-induced overhead

As for the final analysis, the focus shifted from simulating the pick and place application to testing and evaluating the protocol-based interconnection. To evaluate its implementation, it was decided to measure the amount of time needed by GVSoC to perform a specific number of readings on a system's peripheral. Thus, a different, yet simpler, program was realized, containing only the necessary operations. The analysis was performed on three different versions of the architecture: one implementing the original interconnection, a second integrating the delays of the AXI protocol's operations into the functional bus implementation, and a third incorporating the low-level communication channel. The second version of the framework was analyzed to ensure that the implementation of the protocol functioned as expected. In order to compute the delays of the AXI protocol's read and write operations, the number of clock cycles required by each phase was counted and converted into simulation time, knowing that the clock period is $2ms$. The clock cycles are reported in Table 5.2. Eight clock cycles are required to

Operation type	Phase	Actor	#Clock Cycles
Read	**AR**	Manager	3
		Subordinate	3
	R	Manager	1
		Subordinate	1
Write	**AW**	Manager	3
		Subordinate	3
	W	Manager	1
		Subordinate	1
	B	Manager	2
		Subordinate	3

Table 5.2: Number of clock cycles for each phase of a read and write operation

perform a single read operation, totaling $16ms$, while a write operation, on the other hand, requires thirteen clock cycles, totaling $26ms$. The timing information of the operations performed by the three channels are reported in Table 5.3.

The measurements related to the test are shown in Figure 5.4. As expected, it can be seen that the original implementation of the interconnection takes less time

Bus implementation	Operation type	
	Read (ms)	Write (ms)
Functional bus	30	30
Functional bus + AXI delays	46	56
AXI bus	46	56

Table 5.3: Execution times for the read and write operations

to perform the same number of readings as the other two channel implementations, since it does not simulate any low-level exchanges associated with a specific protocol. However, it can be noted that the three curves converge to the same simulation time whenever the number of readings increases to 50000, as the simulated time for that number of readings exceeds the configured limit of a 1000 seconds. The other two curves, instead, are quite similar, presenting negligible time variations for a higher number of readings and demonstrating the correct implementation of the protocol. On average, between the protocol-related curves and the original interconnection curve, a simulation time increase of about 50.49% was measured.

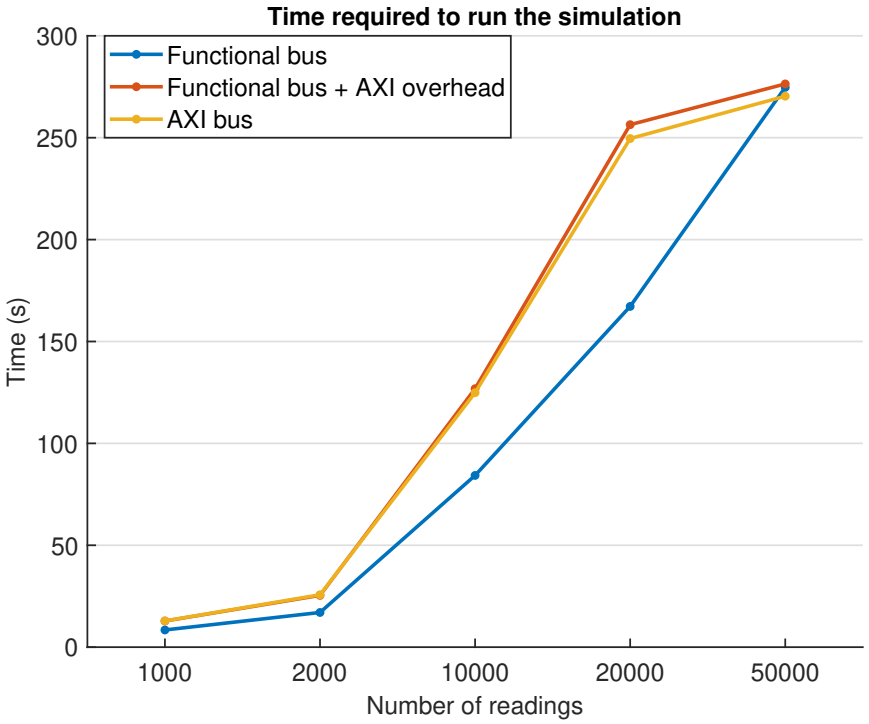


Figure 5.4: Time required by the channels to read a certain number of registers

A variation of this analysis was performed to verify that the AXI channel implementation did not introduce any simulation overhead, aside from further advancing the internal simulated time. To verify the presence of such overhead, the version of the framework containing the high-level implementation of the bus was used to simulate the infinite reading of registers, which was only interrupted upon reaching a specific simulated time, corresponding to the ending time of the other curves represented in Figure 5.4 for a specific number of register accesses. The same measurements as before were taken on the simulator containing the AXI-based interconnection, along with the ending simulated times for each number of readings. The measured ending times are shown in Table 5.4. These values were then used to

# of readings	**1000**	**2000**	**10000**	**20000**
Ending simulated time (ms)	46002	92002	480002	960002

Table 5.4: Ending simulated times for the AXI-based framework

configure the maximum simulation time allowed for simulating the infinite reading of registers performed by the version of the framework containing the high-level

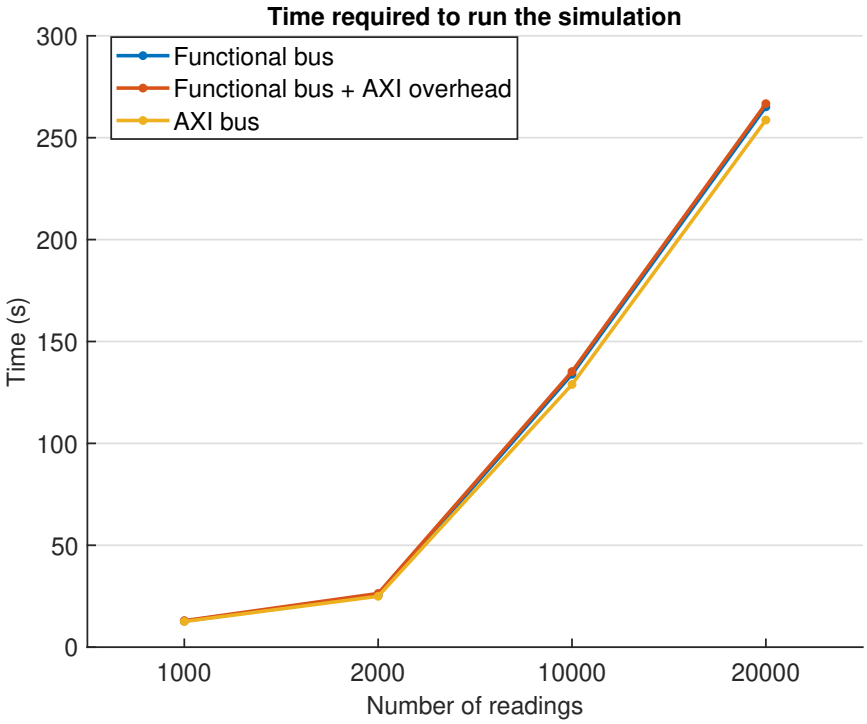


Figure 5.5: Time required by the channels to reach the same simulated time

bus implementation. The results are represented in Figure 5.5, but note that, even though the plot uses the number of readings as the x-axis, the measured values for the version of the framework containing the high-level interconnection refer to the ending times associated with that number of register accesses. The results show that to reach the same simulated time as the protocol-based channels, the functional implementation of the bus requires the same time, demonstrating that no overhead is introduced. Specifically, a difference between the curves exist; however, it consists of about 2.43%, making it negligible.

Chapter 6

Conclusion and Outlook

The aim of this thesis was the integration of a connectivity module and a protocol-based bus implementation into MESSY, a virtual prototyping framework for simulating both functional and extra-functional aspects of an embedded system. The goal of these two additions was to offer a more comprehensive and realistic environment for testing such complex systems. The connectivity module was integrated into the framework using a C-based library named VirtualConnector. This library allows the simulated system's components to connect to an external application and exchange data through a Unix socket structure, simulating aspects that would be difficult to cover with a purely C-based approach. In order to establish communication with any kind of software system, messages are exchanged using a JSON format. On the other hand, the AXI-based interconnection was implemented to integrate a low-level bus channel into the simulator, allowing realistic communication between the system's units. In parallel with its integration into the architecture, the structure of the peripheral components was reworked to further separate custom code implementations from system integration and internal operations, allowing developers to focus on their application-specific requirements. The modified simulator was tested through a pick and place application, where it was connected to Webots and tasked with managing a Franka Emika Panda robotic manipulator. Several measurements were taken, highlighting the importance of simulating low-level exchanges and the difference between manually advancing the simulated time and executing FLOPS operations to reach the same objective. Furthermore, the newly integrated interconnection demonstrated a negligible simulation overhead of about 2% compared to the original version, making it ideal for testing a more realistic environment. Future developments can further expand the work in different directions: a more comprehensive version of the bus protocol can be implemented, adding features to handle multiple manager systems, out-of-order transactions and protection level support; the connectivity module can be extended to support multiple connections and message formats, allowing to connect to more constrained applications; the

example application involving the pick and place simulation within the Webots' environment can be extended to support other robotic manipulators or a more detailed operations' flow.

Appendix A

Listings

```
int ConnectionConfig::initialize_connection() {
    struct sockaddr_un server_address;

    // Create the UNIX socket
    if ((this->server_fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
        return -1;

    // Configure it with the information about the server socket
    memset(&server_address, 0, sizeof(server_address));
    server_address.sun_family = AF_UNIX;
    strncpy(server_address.sun_path, this->socket_path, sizeof(
    ↪ server_address.sun_path)-1);

    // Connect to the server
    if ((this->connection_fd = connect(this->server_fd, (struct
    ↪ sockaddr*) &server_address, sizeof(server_address))) == -1)
        return -1;

    return 0;
}
```

Listing A.1: Socket creation and initialization in MESSY

```
1  json VirtualConnector::read_from_channel(int fd) {
2      int num_read, counter_read, string_length;
3      unsigned char rd_len_buffer[4];
4      char *rd_json_buffer;
5      json data;
6
7      // Read how many characters the json will be long
8      string_length = 0;
9      num_read = read(fd, &rd_len_buffer, 4);
10     for (int i = 0; i < num_read; i++)
11         string_length = string_length | (rd_len_buffer[i] << (8*i));
12
13     // Allocate the memory required to store the json string
14     num_read = 0;
15     counter_read = 0;
16     rd_json_buffer = (char *) malloc(string_length * sizeof(char));
17
18     // Read the message
19     while (counter_read < string_length) {
20         num_read = read(fd, rd_json_buffer + counter_read,
       ↳ string_length - counter_read);
21         counter_read += num_read;
22
23         if (num_read == 0)
24             break;
25     }
26
27     // Convert to JSON
28     data = json::parse(rd_json_buffer);
29     free(rd_json_buffer);
30     return data;
31 }
```

Listing A.2: Read operation from the socket channel in MESSY

```
void VirtualConnector::write_on_channel(int fd, json data) {
    unsigned char length_buffer[4];
    char *wr_buffer;
    int json_length;
    std::string json_string;

    json_string = data.dump();
    json_length = json_string.size();

    // Send the number of bytes
    length_buffer[0] = json_length & 0x000000FF;
    length_buffer[1] = (json_length >> 8) & 0x000000FF;
    length_buffer[2] = (json_length >> 16) & 0x000000FF;
    length_buffer[3] = (json_length >> 24) & 0x000000FF;
    write(fd, length_buffer, sizeof(length_buffer));

    // Write the message on the channel
    wr_buffer = (char *) malloc(json_length * sizeof(char));
    strncpy(wr_buffer, json_string.c_str(), json_length);
    write(fd, wr_buffer, json_length);
    free(wr_buffer);
}
```

Listing A.3: Write operation on the socket channel in MESSY

```
def receive_message(self):
    msg = ""
    byte_read = 0
    msg_length = 0

    # Read how many characters the MESSY json will be long
    msg = self.client_socket.recv(4)
    if msg == b'':
        raise RuntimeError()

    for i in range(4):
        msg_length = msg_length or (msg[i] << (8 * i))

    # Acquire the correct amount of characters to compone the json
    byte_read = 0
    msg = []
    while byte_read < msg_length:
        msg_part = self.client_socket.recv(min(msg_length - byte_read
    ↪ , 4096))
        if msg_part == b'':
            raise RuntimeError()

        byte_read += len(msg_part)
        msg.append(msg_part)

    msg = b''.join(msg).decode("utf-8")
    data = json.loads(msg)
    return data
```

Listing A.4: Read from socket channel in Python

```python
def send_message(self, data):
    msg = bytearray(json.dumps(data), "utf-8") + bytearray(b'\0')
    msg_length = len(msg)

    # Send the length of the json
    byte_sent = self.client_socket.send(msg_length.to_bytes(4, "
    ↪ little"), 4)
    if byte_sent == 0:
        raise RuntimeError()

    # Send the json now
    byte_sent = 0
    total_sent = 0
    while total_sent < msg_length:
        byte_sent = self.client_socket.send(msg[total_sent:])
        if byte_sent == 0:
            raise RuntimeError()

        total_sent += byte_sent

    return
```

Listing A.5: Write on socket channel in Python

```
struct AXI_M_Interface {

    // Clock
    sc_out<bool> clk{"Clock"};

    // Read address channel (AR)
    sc_in<bool> AR_ready{"AR_ready"};
    sc_out<int> AR_address{"AR_Address"};
    sc_out<uint8_t> AR_length{"AR_length"};
    sc_out<bool> AR_valid{"AR_valid"};

    // Read data channel (R)
    sc_in<bool> R_last{"R_Last"};
    sc_in<bool> R_valid{"R_Valid"};
    sc_in<sc_uint<2>> R_resp{"R_resp"};
    sc_in<uint8_t> R_data{"R_Data"};
    sc_out<bool> R_ready{"R_Ready"};

    // Write address channel (AW)
    sc_in<bool> AW_ready{"AW_ready"};
    sc_out<int> AW_address{"AW_Address"};
    sc_out<bool> AW_valid{"AW_valid"};

    // Write data channel (W)
    sc_in<bool> W_ready{"W_Ready"};
    sc_out<uint8_t> W_data{"W_Data"};
    sc_out<bool> W_last{"W_Last"};
    sc_out<bool> W_valid{"W_Valid"};

    // Write response channel (B)
    sc_in<bool> B_valid{"B_valid"};
    sc_in<sc_uint<2>> B_resp{"B_resp"};
    sc_out<bool> B_ready{"B_ready"};
};
```

Listing A.6: AXI manager port signals

```
class Master_2_AXI_Port : virtual public sc_interface {

    public:
        virtual void read(int address, int length,
                          uint8_t* data, uint8_t* response)=0;
        virtual int write(int address, uint8_t* data, int length)=0;
};

class AXI_2_Slave_Port : virtual public sc_interface {

    public:
        virtual bool is_ready()=0;
        virtual void start_transaction()=0;
        virtual void close_transaction()=0;

        virtual bool read(int address, uint8_t* data)=0;
        virtual int write(int address, uint8_t data)=0;
};
```

Listing A.7: Virtual classes definition

Bibliography

[1] Mohamed Amine Hamdi, Giovanni Pollo, Matteo Risso, Germain Haugou, Alessio Burrello, Enrico Macii, Massimo Poncino, Sara Vinco, and Daniele Jahier Pagliari. *Integrating SystemC-AMS Power Modeling with a RISC-V ISS for Virtual Prototyping of Battery-operated Embedded Devices*. 2024. arXiv: 2404.01861 [eess.SY] (cit. on pp. 5, 19, 21).

[2] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. «GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors». In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, Oct. 2021. DOI: 10.1109/iccd53106.2021.00071. URL: http://dx.doi.org/10.1109/ICCD53106.2021.00071 (cit. on pp. 5, 18).

[3] O. Michel. «Webots: Professional Mobile Robot Simulation». In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf (cit. on pp. 5, 21).

[4] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Tech. rep. UCB/EECS-2016-118. May 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html (cit. on p. 6).

[5] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank Gurkaynak, and Luca Benini. *Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices*. Feb. 2017. DOI: 10.1109/TVLSI.2017.2654506. URL: https://ieeexplore.ieee.org/document/7864441 (cit. on p. 6).

[6] Stephan Nolting and All the Awesome Contributors. *The NEORV32 RISC-V Processor*. Aug. 2023. DOI: 10.5281/zenodo.5018888. URL: https://github.com/stnolting/neorv32 (cit. on p. 6).

[7] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (cit. on p. 6).

[8] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. «A Survey on RISC-V-Based Machine Learning Ecosystem». In: *Information* 14.2 (2023). ISSN: 2078-2489. DOI: 10.3390/info14020064. URL: https://www.mdpi.com/2078-2489/14/2/64 (cit. on p. 6).

[9] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. «A comparative survey of open-source application-class RISC-V processor implementations». In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. CF '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 12–20. ISBN: 9781450384049. DOI: 10.1145/3457388.3458657. URL: https://doi.org/10.1145/3457388.3458657 (cit. on p. 6).

[10] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. «Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing». In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: 10.1109/JSSC.2019.2912307 (cit. on p. 6).

[11] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. «Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX». In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145 (cit. on p. 6).

[12] *PULPino - An open-source microcontroller system based on RISC-V*. https://github.com/pulp-platform/pulpino. Accessed: August 1, 2024 (cit. on p. 6).

[13] Jackrit Suthakorn. «A concept on Cooperative Tele-Surgical System based on Image-Guiding and robotic technology». In: *2012 Pan American Health Care Exchanges*. 2012, pp. 41–45. DOI: 10.1109/PAHCE.2012.6233437 (cit. on p. 6).

[14] Cuebong Wong, Erfu Yang, Xiu-Tian Yan, and Dongbing Gu. «An overview of robotics and autonomous systems for harsh environments». In: *2017 23rd International Conference on Automation and Computing (ICAC)*. 2017, pp. 1–6. DOI: 10.23919/IConAC.2017.8082020 (cit. on p. 6).

[15] Selim Solmaz et al. «Robust Robotic Search and Rescue in Harsh Environments: An Example and Open Challenges». In: *2024 IEEE International Symposium on Robotic and Sensors Environments (ROSE)*. 2024, pp. 1–8. DOI: 10.1109/ROSE62198.2024.10591144 (cit. on p. 6).

[16] Rohan Thakker, Ajinkya Kamat, Sachin Bharambe, Shital Chiddarwar, and K. M. Bhurchandi. «ReBiS - Reconfigurable Bipedal Snake robot». In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 309–314. DOI: 10.1109/IROS.2014.6942577 (cit. on p. 7).

[17] Lillian Chin, Max Burns, Gregory Xie, and Daniela Rus. «Flipper-Style Locomotion Through Strong Expanding Modular Robots». In: *IEEE Robotics and Automation Letters* 8.2 (2023), pp. 528–535. DOI: 10.1109/LRA.2022.3227872 (cit. on p. 7).

[18] A.S. Mohamed Sahan, S. Kathiravan, M. Lokesh, and R. Raffik. «Role of Cobots over Industrial Robots in Industry 5.0: A Review». In: *2023 2nd International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*. 2023, pp. 1–5. DOI: 10.1109/ICAECA56562.2023.10201199 (cit. on p. 8).

[19] Munawar A. Riyadi, Norman Sudira, M. H. Hanif, and Aris Triwiyatno. «Design of pick and place robot with identification and classification object based on RFID using STM32VLDISCOVERY». In: *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*. 2017, pp. 171–176. DOI: 10.1109/ICECOS.2017.8167128 (cit. on p. 8).

[20] Alberto Borboni, Karna Vishnu Vardhana Reddy, Irraivan Elamvazuthi, Maged Al-Quraishi, Elango Natarajan, and Syed Saad Azhar Ali. «The Expanding Role of Artificial Intelligence in Collaborative Robots for Industrial Applications: A Systematic Review of Recent Works». In: *Machines* 11 (Jan. 2023), p. 111. DOI: 10.3390/machines11010111 (cit. on p. 10).

[21] Giorgia Chiriatti, Giacomo Palmieri, Cecilia Scoccia, Matteo Claudio Palpacelli, and Massimo Callegari. «Adaptive Obstacle Avoidance for a Class of Collaborative Robots». In: *Machines* 9.6 (2021). ISSN: 2075-1702. DOI: 10.3390/machines9060113. URL: https://www.mdpi.com/2075-1702/9/6/113 (cit. on p. 10).

[22] Peter Corke and Jesse Haviland. «Not your grandmother's toolbox–the Robotics Toolbox reinvented for Python». In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 11357–11363 (cit. on pp. 10, 29).

[23] *The Franka Emika Panda manipulator*. https://ifdesign.com/en/winner-ranking/project/franka-emika-panda/253653. Accessed: July 25, 2024 (cit. on p. 11).

[24] Amit Rogel, Richard Savery, Ning Yang, and Gil Weinberg. «RoboGroove: Creating Fluid Motion for Dancing Robotic Arms». In: June 2022, pp. 1–9. DOI: 10.1145/3537972.3537985 (cit. on p. 11).

[25] *Learn the architecture - An introduction to AMBA AXI*. Arm Limited. 2022 (cit. on pp. 14–17).

[26] «IEEE Standard for Standard SystemC® Language Reference Manual». In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), pp. 1–618. DOI: 10.1109/IEEESTD.2023.10246125 (cit. on p. 20).

[27] «IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual». In: *IEEE Std 1666.1-2016* (2016), pp. 1–236. DOI: 10.1109/IEEESTD.2016.7448795 (cit. on p. 20).

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. «ROS: an open-source Robot Operating System». In: vol. 3. Jan. 2009 (cit. on p. 22).

[29] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 22).

[30] PULP Platform. *AXI SystemVerilog Modules for High-Performance On-Chip Communication*. Version 0.39.4. July 2024. URL: https://github.com/pulp-platform/axi (cit. on p. 22).

[31] Alex Forencich. *Verilog AXI Components*. Accessed: August 14, 2024. URL: https://github.com/alexforencich/verilog-axi (cit. on p. 22).

[32] MINRES Technologies. *SystemC-Components (SCC)*. Version 2024.07. July 2024. URL: https://github.com/Minres/SystemC-Components (cit. on p. 22).

[33] Leonard Lochte-Holtgreven. *GAPBOTS: Combining System-on-Chip and Nano-UAV Simulations*. Master's thesis. Zurich, Jan. 2024 (cit. on p. 23).

[34] Niels Lohmann. *JSON for Modern C++*. Version 3.11.3. Nov. 2023. URL: https://github.com/nlohmann (cit. on p. 28).

[35] Yanhao He and Steven Liu. «Analytical Inverse Kinematics for Franka Emika Panda – a Geometrical Solver for 7-DOF Manipulators with Unconventional Design». In: *2021 9th International Conference on Control, Mechatronics and Automation (ICCMA)*. 2021, pp. 194–199. DOI: 10.1109/ICCMA54375.2021.9646185 (cit. on p. 29).

[36] Sven Tittel. «Analytical Solution for the Inverse Kinematics Problem of the Franka Emika Panda Seven-DOF Light-Weight Robot Arm». In: *2021 20th International Conference on Advanced Robotics (ICAR)*. 2021, pp. 1042–1047. DOI: 10.1109/ICAR53236.2021.9659393 (cit. on p. 29).

[37] Edward Humes, Mozhgan Navardi, and Tinoosh Mohsenin. *Squeezed Edge YOLO: Onboard Object Detection on Edge Devices*. 2023. arXiv: 2312.11716 [cs.CV]. URL: https://arxiv.org/abs/2312.11716 (cit. on p. 49).