TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Dissertation

Reconfigurable Computing Systems for Robotics using a Component-Oriented Approach

Ariel Podlubne

Born on: 29th May 1987 in Salta, Argentina
Matriculation year: 2017

to achieve the academic degree

Doktor-Ingenieur (Dr.-Ing.)

Supervisor and examiner
Prof. Dr.-Ing. Diana Göhringer

Co-examiner
Prof. Dr. Pedro Diniz

Submitted on: 4th May 2023
Defended on: 28th June 2023

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Statement of authorship

I hereby certify that I have authored this document entitled *Reconfigurable Computing Systems for Robotics using a Component-Oriented Approach* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Diana Göhringer

Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 4th May 2023

Ariel Podlubne

Abstract

Robotic platforms are becoming more complex due to the wide range of modern applications, including multiple heterogeneous sensors and actuators. In order to comply with real-time and power-consumption constraints, these systems need to process a large amount of heterogeneous data from multiple sensors and take action (via actuators), which represents a problem as the resources of these systems have limitations in memory storage, bandwidth, and computational power.

Field Programmable Gate Arrays (FPGAs) are programmable logic devices that offer high-speed parallel processing. FPGAs are particularly well-suited for applications that require real-time processing, high bandwidth, and low latency. One of the fundamental advantages of FPGAs is their flexibility in designing hardware tailored to specific needs, making them adaptable to a wide range of applications. They can be programmed to pre-process data close to sensors, which reduces the amount of data that needs to be transferred to other computing resources, improving overall system efficiency. Additionally, the reprogrammability of FPGAs enables them to be repurposed for different applications, providing a cost-effective solution that needs to adapt quickly to changing demands. FPGAs' performance per watt is close to that of Application-Specific Integrated Circuits (ASICs), with the added advantage of being reprogrammable.

Despite all the advantages of FPGAs (e.g., energy efficiency, computing capabilities), the robotics community has not fully included them so far as part of their systems for several reasons. First, designing FPGA-based solutions requires hardware knowledge and longer development times as their programmability is more challenging than Central Processing Units (CPUs) or Graphics Processing Units (GPUs). Second, porting a robotics application (or parts of it) from software to an accelerator requires adequate interfaces between software and FPGAs. Third, the robotics workflow is already complex on its own, combining several fields such as mechanics, electronics, and software.

There have been partial contributions in the state-of-the-art for FPGAs as part of robotics systems. However, a study of FPGAs as a whole for robotics systems is missing in the literature, which is the primary goal of this dissertation. Three main objectives have been established to accomplish this. (1) Define all components required for an FPGAs-based system for robotics applications as a whole. (2) Establish how all the defined components are related. (3) With the help of Model-Driven Engineering (MDE) techniques, generate these components, deploy them, and integrate them into existing solutions.

The component-oriented approach proposed in this dissertation provides a proper solution for designing and implementing FPGA-based designs for robotics applications. The modular architecture, the tool "FPGA Interfaces for Robotics Middlewares" (FIRM), and the toolchain "FPGA Architectures for Robotics" (FAR) provide a set of tools and a comprehensive design process that enables the development of complex FPGA-based designs more straightforwardly and efficiently. The component-oriented approach contributed to the state-of-the-art in FPGA-based designs significantly for robotics applications and helps to promote their wider adoption and use by specialists with little FPGA knowledge.

Acknowledgment

I would like to express my sincere gratitude to Prof. Diana Göhringer for providing me with the opportunity to join the Chair of Adaptive Dynamic Systems. I am thankful for the open door that was meant for “five minutes” but often extended into hour-long discussions, even when they took unexpected turns and led us into discussions completely unrelated to the original topic. Your guidance over the years, with precise redirections and the freedom to explore different ideas, has been invaluable.

My deepest appreciation goes to Johannes Mey, marking the longest collaboration of my Ph.D. journey[1]. Thank you for the countless discussions, collaborative problem-solving, and the dedicated hours that shaped this research. I extend my gratitude to Sergio Pertuz for joining Johannes and me towards the end of our work. I look forward to seeing the positive impact of our collaborative work in many years to come.

Special thanks to René Schöne for his critical perspectives, which have significantly enhanced the rigor of this research, and to my office mate, Ahmed Kamaledin, for the endless conversations and the shared time.

Lastly, I am profoundly grateful for my family, who consistently supports me in both good times and, especially, when things do not go as expected.

[1]Work funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany’s Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence “Centre for Tactile Internet with Human-in-the-Loop” (CeTI) of Technische Universität Dresden.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit.
AST	Abstract Syntax Tree.
AXIS	AXI Stream.
BNF	Backus-Naur Form.
CAD	Computed Aided Design.
CFG	Context-Free Grammar.
CLB	Configurable Logic Block.
CPU	Central Processing Unit.
DDS	Data Distribution Service.
DMA	Direct Memory Access.
DNN	Deep Neural Network.
DoF	Degrees of Freedom.
DPR	Dynamic Partial Reconfiguration.
DSE	Design Space Exploration.
DSL	Domain Specific Language.
DSP	Digital Signal Processor.
DUT	Device Under Test.
EDF	Earliest Deadline Frist.
FAR	FPGA Architectures for Robotics.
FF	Flip-Flop.
FIFO	First In First Out.
FIRM	FPGA Interfaces for Robotics Middlewares.
FP	Fixed Priority.
FPGA	Field Programmable Gate Array.

FPS	Frames per Second.
FSM	Finite State Machine.
GPU	Graphics Processing Unit.
GUI	Graphical User Interface.
HDL	Hardware Description Language.
HDT	Hardware Description Template.
HLS	High-Level Synthesis.
IDL	Interface Definition Languages.
IoT	Internet of Things.
IP	Intellectual Property.
LHS	Left-Hand-Side.
LoC	Lines of Code.
LOEDF	Latency-Optimized Earliest Deadline First.
LRU	Least Recently Used.
LST	Least Slack Time.
LUT	Lookup Table.
MARTE	Modeling and Analysis of Real-Time and Embedded Systems.
MDA	Model-Driven Architecture.
MDE	Model-Driven Engineering.
NoC	Network-on-Chip.
NPFP	Non-Preemptive Fixed Priority.
NTA	Non-Terminal Attribute.
OS	Operating System.
PE	Processing Element.
PFP	Preemptive Fixed Priority.
PIM	Platform Independent Model.
PL	Programmable Logic.
PS	Processing System.
PSM	Platform Specific Model.
RAG	Reference Attribute Grammar.
RCS	Reconfigurable Computing System.
RHS	Right-Hand-Side.
ROEDF	Resource-Optimized Earliest Deadline First.

ROS	Robot Operating System.
RPC	Remote Procedure Call.
RTL	Register Transfer Level.
SIMD	Single Instruction Multiple Data.
SLAM	Simultaneous Localization and Mapping.
SoC	System-on-Chip.
SPI	Serial Peripheral Interface.
UML	Unified Modeling Language.
VHDL	Very High Speed Hardware Description Language.
WCET	Worst Case Execution Time.
YAML	Yet Another Markup Language.

1 Introduction

1.1 Motivation

Robotics has become an important field over the last decades in the research community as well as in industry, but there are still open challenges to solve, such as new fabrication schemes, new power sources, battery technology, and energy-harvesting schemes, navigation in extreme environments or artificial intelligence for robotics [1]. The application fields range from manufacturing [2], collaborative robots (cobots) interacting directly with humans [3], biomedicine [4], drones for different application [5] as well as mobile robots [6], to name a few. Due to the wide range of applications, robotic platforms are becoming more complex, including heterogeneous sensors and actuators. Complexity increases even more, when multiple robots are part of the same system, such as an automated warehouse [7]. All these systems need to process a large amount of heterogeneous raw data from multiple sensors and take action (via actuators), complying with real-time and power-consumption constraints. However, these systems usually have limited resources in terms of memory storage, bandwidth, and computational capabilities.

On the one hand, Central Processing Units (CPUs) have been traditionally the defacto Processing Element (PE) as they can handle a wide range of tasks quickly, and there is much support concerning their programmability. However, even though they include multiple cores, they are limited in terms of running many tasks in parallel. On the other hand, heterogeneous computing has grown over the last years, improving the innovations on accelerating compute-intensive workloads such as artificial intelligence [8]. The field of computer architecture has become quite diverse with the emergence and constant improvements of CPUs, Digital Signal Processors (DSPs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Lately, these last two have been explored as PEs for robotics.

GPUs include a large number of processing cores designed to run simultaneously, enabling a vast level of parallelism. In terms of their programmability, different frameworks have been used by developers. The parallel programming paradigm Compute Unified Device Architecture (CUDA®) was released by NVIDIA® in 2007. It is very similar to the C language, oriented to GPUs. It combines serial and parallel executions and contains a particular C function (kernel) executed concurrently on a fixed number of threads. The Open Computing Language (OpenCL™) was launched to provide benchmarks for heterogeneous computing. It offers a portable language for GPUs and is used to design applications that are general enough to run on different architectures.

FPGAs are ideal candidates to process a large amount of heterogeneous data due to their intrinsic parallel architecture [9]. They provide versatility to design hardware according to the needs precisely, can pre-process data very close to sensors [10], and can be, from the performance per watt, close to Application-Specific Integrated Circuits (ASICs) [11, 12], but they are more flexible as they are reprogrammable. However, their programmability is not as easy as CPUs or GPUs. A similar approach to CUDA® and OpenCL™ is High-Level Synthesis (HLS), which is also a C-like design process in which a high-level functional description of a design can be compiled into Register Transfer Level (RTL). It allows designers with basic hardware knowledge to re-use software applications with minimal changes to comply with some hardware constraints, such as data movement between different components (calling functions in terms of software). Therefore, FPGAs are not limited anymore to experienced hardware developers with knowledge in Hardware Description Languages (HDLs). Nevertheless, designers still need some basic hardware knowledge to consider when "coding" new accelerators (also called Intellectual Property (IP) cores, hardware IPs, or just IPs). The flexibility of FPGAs is due to the programmability of their Configurable Logic Blocks (CLBs) and the interconnection among them. Besides, FPGAs's programmable connections to external components make them very versatile for systems with many sensors and actuators, whether existing or new ones that may be required. These last two points are the main reasons why FPGAs are becoming an ideal candidate to be used as a computational element in robotic applications. The challenge with FPGAs, as with GPUs, is how to integrate them into a given system or architecture, which is the main motivation of this dissertation. It is important to note that the embedded systems community has also focused its attention towards FPGAs [13], bringing new tools and programming paradigms, primarily based on HLS. Different commercial (e.g., Xilinx® OpenCV, Matlab® HDL Coder), as well as academic frameworks [14, 15] are available, providing multiple functions as individual elements (e.g., filters) to be integrated into a given architecture. Lastly, they are a good fit to *self-adaptive systems*, as some blocks can be modified dynamically at runtime.

Robotic platforms are a combination of software and hardware, requiring specific knowledge of multiple fields (i.e., hardware, software, control). Extending their capabilities with different computing systems increases their complexity even further. Ideally, experts in each field focus on a specific system part according to their expertise in developing robots. They should complement each other, and designers should be provided with simple tools to focus mainly on their expertise. However, there are challenges in coping with these sophisticated heterogeneous systems and how to integrate them.

Regarding software, the robotics community adopted the Robot Operating System (ROS) [16] as the mainstream middleware over the last years. Figure 1.1 shows the total number of packages downloaded over the last decade, which include standard algorithms for basic tasks such as localization, control, or mapping, to name a few, freely available in ROS due to its large community behind it. Lately, efforts have been put into ROS2 to improve its real-time features and safety-critical related systems and ROS-industrial to extend the capabilities of ROS software to industrial relevant hardware and applications. All of these lead to more designers working on complex robotic systems, demanding more computational power and often needing to process a large amount of data in parallel.

One aspect to remember in software development is that CPUs have been normally designed to fulfill generic operations such as addition or subtraction. Therefore, in order to obtain any given result, multiple operations have to be performed. A second aspect is that complex

[1] `http://wiki.ros.org/Metrics`

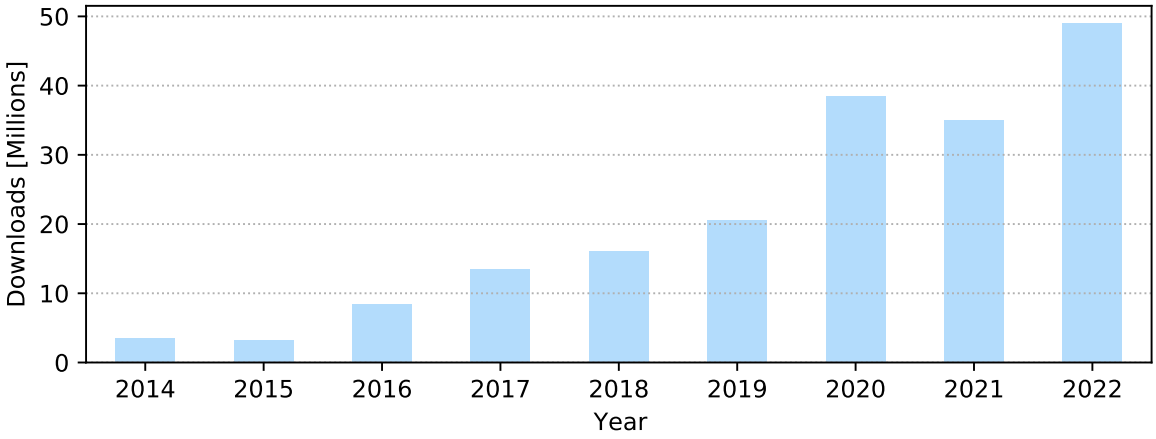


Figure 1.1: Total ROS packages downloaded (data based on ROS Community Metrics[1])

operations can be achieved by designing specific hardware. For example, multiple functions for image processing can be implemented in an FPGAs, combined and produce powerful algorithms [17]. A third aspect concerns that these complex robots have to perform many computations. Reaching the desired performance represents a challenge, especially in the case when they are equipped with embedded computers consuming a lot of energy, which is limited in a mobile system that is usually battery operated. Therefore, other computational resources, such as FPGAs or GPUs, which are more power efficient, should be considered for robotics. The first one is advantageous due to their versatility in obtaining custom designs capable of performing heavy computations with reduced power consumption. Moreover, they have been proven to be more efficient in terms of energy compared to general purpose processors [18]. Besides, FPGAs handle parallel and logic operations better than CPUs.

Despite all these and the advantages of FPGAs mentioned before, the robotics community has not fully included them so far as part of their systems for several reasons. First, designing FPGA-based solutions requires hardware knowledge and longer development times than software solutions. Second, porting a robotics application (or parts of it) from software to an accelerator requires adequate interfaces between software and FPGAs. Third, the robotics workflow is already complex on its own, combining several fields such as mechanics, electronics, and software. Hence, increasing the effort to develop these systems is not desired [19, 20]. Lastly, there is a knowledge gap in system integration besides significant design engineering costs, which is detrimental to integrating new Reconfigurable Computing Systems (RCSs) into robotics.

The main four points to take into account when considering RCSs for robotics are:

1. The hardware platform must be able to comply with the power, computing and energy consumption requirements needed for robotic applications. They should be flexible and adaptable for reusability, and provide simple programmability.

2. Traditional robotics systems are software-based, so there has to be an easy integration of existing architectures to a new platform to enhance their capabilities [21].

3. The usability of such new platforms must be easily adapted by roboticists to attract them to use such systems.

4. Developing FPGA-based architectures and system integration is a complex and arduous process that is usually overlooked, the generation of all components needed and their complete deployment should ideally not require much manual intervention.

These points already involve too many aspects to consider, so a simplification is required, at least from a design point of view. A model is an abstraction of a system or the real world concentrating on specific structural or behavioral properties and representing them in a syntactically and semantically defined language [22]. Systems and, therefore, robots can be developed and tested based on such models exploiting their ability to abstract. By abstracting some details of a system in its model, the complexity of the modeled system is hidden, thus enhancing the understanding of the system [23]. In this respect, a suitable level of abstraction and an appropriate system view must be determined. A too detailed model may not only be limited to a single use case but also be hard to construct because of time constraints or high complexity [24]. However, a too generic model may not provide the required expressiveness. Thus, the selection of the right level of abstraction is essential.

Figure 1.2 shows an overview of the *objectives and contributions* of this dissertation. The figure depicts the three main aspects to consider and the solutions proposed to achieve said objectives. Based on the motivation and background in this section, in the following one, existing research challenges are discussed, shaping the objective of this work.

1.2 Objectives

There have been partial contributions in the state-of-the-art for RCSs, particularly FPGAs, as part of robotics systems [27]. However, a study of FPGAs as a whole for robotics systems is missing in the literature. This means that defining which are all the components required for an FPGA-based system for robotics applications as a whole, their integration into existing solutions as well as the generation of said components has not been done, which are the main objectives of this dissertation. In order to achieve this, the following points have been defined, as they have not been done so far, making them the three main objectives for this dissertation:

1. ***Objective 1***: Define *all* components required for an FPGA-based system for robotics applications as a whole.
2. ***Objective 2***: Establish how all the defined components are related.
3. ***Objective 3***: The generation of these components, their deployment and integration into existing solutions.

All along this work, Xilinx® has been the FPGA reference, particularly the Zynq® model, which includes processors, called the Processing System (PS) and the logic part, referred to as Programmable Logic (PL). Note that the concepts described in this work are not only valid for Xilinx® FPGAs as one could also think of soft-core processors such as MicroBlaze™ or RISC-V cores.

The following section describes the contributions proposed to meet the objectives described previously.

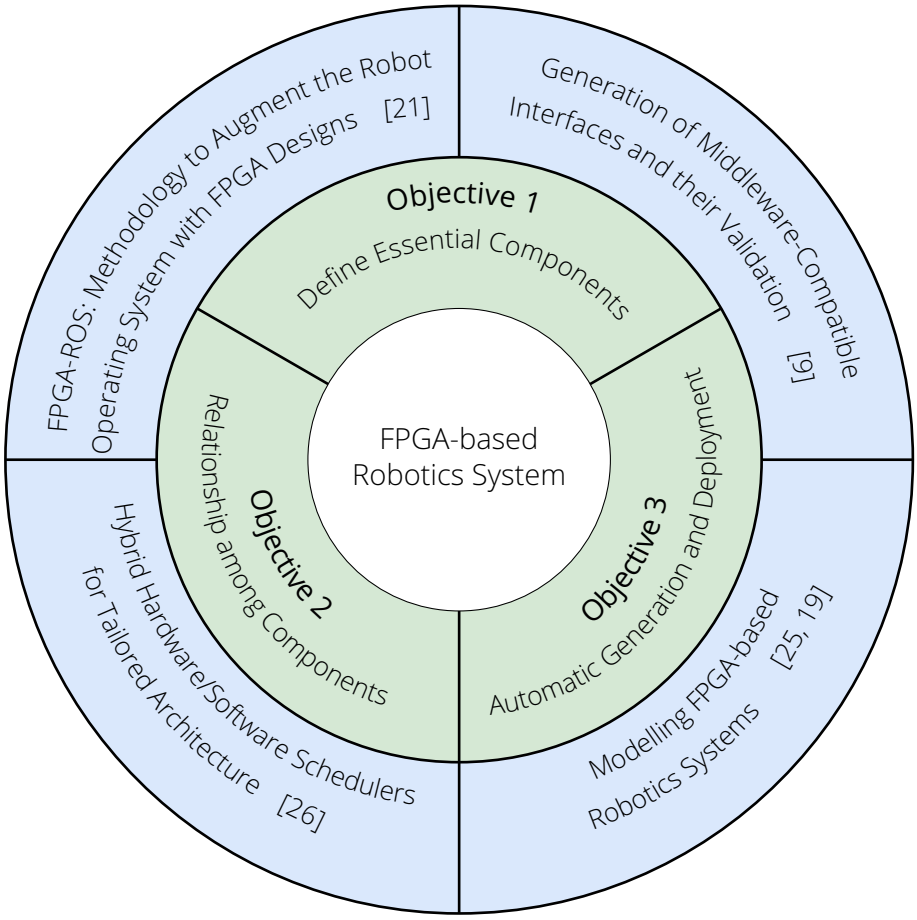


Figure 1.2: Objectives and contributions

1.3 Contributions

The main contributions of this dissertation are presented below:

- **FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs** [21]
  The first contribution of this dissertation is the foundation of the work. It concerns a *methodology* to design custom FPGA-based architectures compatible with the mainstream robotics middleware ROS. The aim is for highly computational algorithms implemented as dedicated hardware modules to increase the processing power of any heterogeneous robotics system, taking advantage of the freedom and versatility that FPGAs provide. A modular design is foreseen to ease the adaptability to changes in the system. The selection of the AXI Stream (AXIS) protocol enables modules to be added or removed dynamically ("plug & play"). Furthermore, they can be designed in HDL (e.g., Very High Speed Hardware Description Language (VHDL), Verilog) or HLS. Message specifications (either off-the-shelf or custom ones) representing data structures for the entities of accelerators are used. Converters act as encoders/decoders for the IP cores and the common AXIS interface to communicate with other modules in the architecture.

- Hybrid Hardware/Software Schedulers for Tailored Architecture [26]
  The *base architecture* is generic so that it can host any number of accelerators, whether they exchange data among them, receive data from the PS or send data to the PL. Therefore, how this communication is established has to be addressed. On the software side, where the native middleware runs, the goal is to share the most up-to-date incoming data with accelerators. On the hardware side, the main goal is to serve all accelerators in the system and avoid them being unable to send or receive data. Six different schedulers are proposed to cover multiple scenarios for different robotics applications. They are scalable and easy to adapt to manage either a small or large number of accelerators. Furthermore, the evaluation framework can also be used to select the most fitted algorithm for each application, depending on the total number of accelerators and their characteristics.

- Generation of Middleware-Compatible Interfaces and their Validation [9]
  The *base architecture* needs suitable interfaces compatible with robotic applications. They must be compatible with middleware specifications, which describe data types and structures to transfer information from/to different parts of the architecture. These specifications are what define the *converters*. Their hardware implementation deals with low-level details, usually abstracted in the software workflow. Therefore, writing these converters manually is a cumbersome and error-prone process. On the one hand, a model-based toolchain that automatically generates these hardware components (VHDL modules) from existing message specifications is proposed. On the other hand, the model-based approach allows for validating their *correct logic*. Lastly, the approach facilitated the extension from ROS1 to easily provide support for ROS2, and other middlewares can also be incorporated.

- Modelling FPGA-based Robotic Systems [25, 19]
  The *system integration* of all generated components to deploy complex systems compliant with existing robotics middlewares (e.g., ROS, ROS2) is an arduous and error-prone process. Therefore, a modeling approach to solve this is proposed. As the aim is not to increase the workflow of roboticists, the way the system is specified must be compact yet expressive with just enough information to generate all required components and to integrate existing algorithms. The proposed approach (c.f., Figure 1.3) exploits the advantages of Model-Driven Engineering (MDE) and model-based code generation to produce all components. Data type and data flow analysis are performed to derive the necessary information to generate the components and their connections.

An overview of the process, including all contributions listed above, is presented in Figure 1.3 [19]. It is based on the MDE technique [28], which uses a staged model transformation process in which models are transformed in iterations. To separate the resulting logical structure of the code from the concrete syntax, a logic-less (i.e., containing no complex template expansion logic) template engine is used (*Generators* in Figure 1.3). Their inputs are specifications of the component they will generate, an intermediate model (template configuration) containing all the information to generate the tailored artifacts (i.e., VHDL, C++ files for HLS, TCL or bash scripts) and their corresponding template. The workflow can be split into three branches. The first refers to the *base architecture*, which includes two distinct *generators*. One for the *architecture-dependent* components, like a *manager* (which includes the scheduler) to handle N accelerators and a *communication interface* to exchange data with external non-hardware components. The other one handles the components of the *processor-related* (hard or soft processors). The second branch handles the *interfaces to robotic*

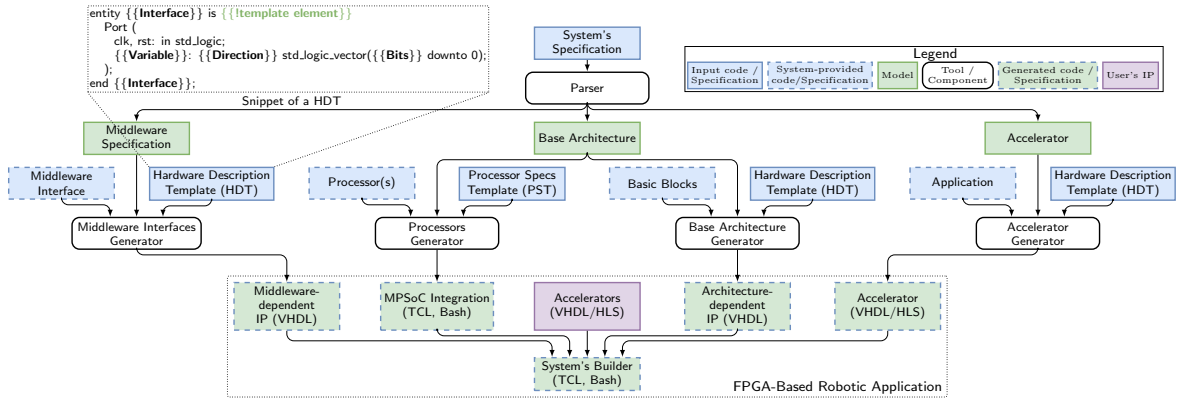


Figure 1.3: Component-oriented workflow for the generation of FPGA-based robotic applications

applications. The last branch generates *accelerators* based on the expected behavior for a given application, compatible with middleware specifications (interfaces). The output of all of them is a set of artifacts for all components needed to automatically deploy an *FPGA-based robotics application* from a simple system's specification.

1.4 Thesis Structure

This dissertation is structured in six chapters, including this one. An overview of each of them is given below.

Chapter 2 presents the background and state-of-the-art review of the related topics of this dissertation. Three main points are discussed, which complement each other, to pave the way for the contributions presented previously. The first one explores traditional (non-hardware-oriented) methodologies followed in robotics. Then, an overview is given about which applications in the robotics field use GPUs and FPGAs to understand their complexity, advantages, and challenges they bring. Furthermore, they depict why especially FPGAs are a good fit for robotics applications. Lastly, an analysis of different modeling techniques to tackle the complexity of these applications and how to circumvent them by code generation is presented.

Chapter 3 introduces the base architecture, discusses why a modular approach is needed, the reason for each component to be part of the system, how they relate to each other, and how each of them interact with one another. Then, it presents the details of all the components that constitute it, analyzing their scalability and generalizability to multiple use cases.

Chapter 4 presents several scheduler algorithms to arbitrate the exchange of data between software and hardware components. Experimental results are presented, describing all the characteristics that can be used to select the most appropriate one for each application.

Chapter 5 addresses the generation of hardware interfaces compatible with standard robotics systems, based on specifications. The chapter starts with the goals and challenges, followed by a description of the toolchain proposed. The fundaments of the model-based approach

chosen and why a logicless template engine is chosen are described. The underlying reason for the need for a toolchain is discussed based on the complexities of message specifications. Furthermore, the full ROS support and the means to extend the toolchain for ROS2 are explained. An evaluation of the toolchain follows to show the correctness of the logic of the generated components and the claimed full ROS support.

Chapter 6 describes the extended toolchain to generate the entire FPGA-based system in addition to the hardware interfaces. It starts with the requirements on how to describe the system in a compact yet power expressive [29] way to obtain the explicit and implicit characteristics of the system. Then, a description of the system's model is illustrated, which is used to generate all components. Three challenges faced in the generation of such architecture are presented and how they are solved with the proposed approach. Lastly, an evaluation with several use cases is performed, followed by comparing the effort to write them manually versus using the resulting toolchain to highlight its benefits.

Chapter 7 summarizes and concludes this dissertation and presents the future work.

2 Background and State-of-the-Art

In recent years, robotics research has witnessed remarkable advancements in various applications, ranging from manufacturing, healthcare, and logistics, to name a few. These applications require powerful computational systems that can process large amounts of heterogeneous data from multiple sensors fast, usually at real-time, to enable quick and precise decisions that trigger appropriate actions on different actuators. Despite significant progress in robotics technology, the challenge of processing and integrating data from diverse sources in real-time remains a significant issue that researchers still need to tackle. Furthermore, new PEs have been emerging over the last years. However, but their integration into the current robotics workflow tends to be an arduous process, leaving them out of consideration by roboticists due to their difficulties in regards to usability and programmability. Efficient robotic systems require innovative solutions that can handle large-scale, complex data processing tasks. These solutions should be robust, reliable, and scalable, and easy to integrate into existing systems.

All these systems need to process a large amount of heterogeneous raw data from multiple sensors and take action (via actuators), complying with real-time and power-consumption constraints. However, these systems usually have limited resources in terms of memory storage, bandwidth, and computational capabilities. Traditionally, CPUs have been the defacto PE as they can handle a wide range of tasks quickly, and there is much support concerning their programmability. However, even though they include multiple cores, they are limited in terms of running many tasks in parallel. Lately, two other options have been explored as PEs, namely GPUs and FPGAs.

GPUs include a large number of processing cores designed to run simultaneously, enabling a vast level of parallelism. In terms of their programmability, different frameworks have been used by developers. The parallel computing platform and application programming interface Compute Unified Device Architecture (CUDA) was released by NVIDIA in 2007. It is very similar to the C language oriented to GPUs. It combines serial and parallel executions and contains a particular C function (kernel) executed concurrently on a fixed number of threads. The Open Computing Language (OpenCL) was launched to provide benchmarks for heterogeneous computing. It offers a portable language for GPUs and is used to design applications that are general enough to run on different architectures.

FPGAs are ideal candidates to process a large amount of heterogeneous data due to their intrinsic parallel architecture [9]. They provide versatility to design hardware according to the needs precisely and can pre-process data very close to sensors [10]. Moreover, they provide better performance per Watt than a standard CPU-based architecture [30]. However, their programmability is not as easy as CPUs or GPUs. A similar approach to CUDA and OpenCL is

HLS, which is also a C-like design process in which a high-level functional description of a design can be compiled into RTL. Nevertheless, designers still need some basic hardware knowledge to take into account some details when “coding” new accelerators (also called hardware IPs). FPGAs offer a better energy efficiency compared to CPUs and GPUs [18], becoming an ideal candidate to be used as a computational element in robotic applications.

Despite the advantages of FPGAs and GPUs, their programmability and integration as individual modules into existing systems are cumbersome. An efficient option to aid the integration of multiple components are middlewares, which provide services beyond those available from the Operating System (OS). They make it easier for developers to implement communication and input/output to focus on the application’s specific purpose. Nevertheless, there are plenty from the software side. However, only a little support from the hardware side focuses on accelerators, so the experience is needed to incorporate them into existing solutions. To simplify the process, a higher level of abstraction helps, which is why model-based approaches have been proposed to integrate, generate and deploy hybrid software/hardware systems.

Robotic systems require specific knowledge of multiple fields (i.e., hardware, software, control). Extending their capabilities with different computing systems increases their complexity even further. Figure 2.1 shows data from peer-reviewed articles, depicting how GPUs and FPGAs have been part of robotic-related research, which *has not* been increasing over the last ten years. Furthermore, Figure 2.1 also shows in terms of percentage, how many publications related to robotics (that include the word *robot*) are concerned with *FPGA* or *GPU*. It can be observed that FPGAs have not been widely adopted as an alternative PEs in robotics since the early 2010s. In recent years, GPUs have gained popularity, most likely due to the increasing affordability of embedded GPUs. This represents one of the primary motivations for this dissertation, which aims to investigate the underlying reasons for the limited use of FPGAs in robotics and propose methodologies and tools for enhancing their adoption as a viable PE option in this field.

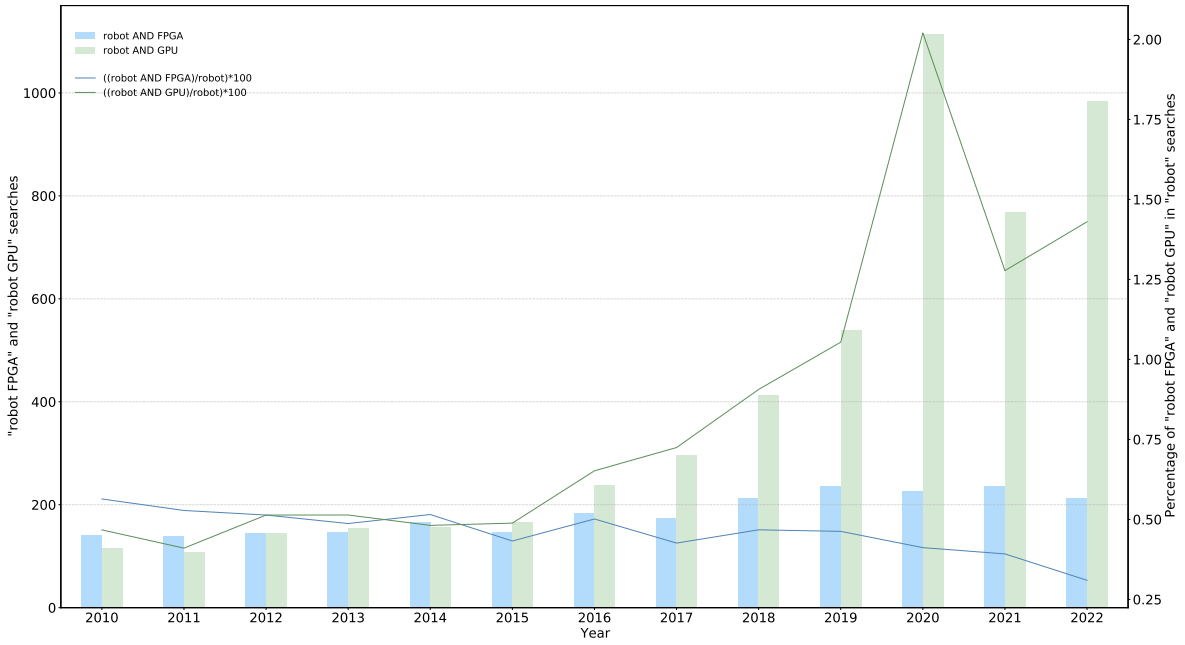


Figure 2.1: Related FPGA and GPU publications for robotic applications.

Ideally, to develop robots, experts in each field focus on a specific system part according to their expertise. However, there are challenges in coping with these sophisticated heterogeneous systems and how to integrate them. MDE copes with the challenges of building complex heterogeneous systems [28]. A model can be defined as an *abstraction* of a system often used to replace the system under study [31]. They represent a partial and simplified view of a system. So, modeling all parts of the system separately is usually necessary to better represent and understand the system under study [31].

It is important to define some concepts that will be used for the remaining of this manuscript, before exploring the state-of-the-art. These are introduced in Sections 2.1 and 2.3 to 2.6.

2.1 Zynq and UltraScale FPGA Families

This section introduces the main concepts about FPGAs followed in this dissertation. Figure 2.2 show a generic representation of FPGAs which helps to understand why they are considered an ideal candidate for robotics systems in this work. First of all, they include CLBs, which are the essential resources FPGAs include. They usually include Flip-Flops (FFs), Lookup Tables (LUTs) and multiplexer, enabling the programmability mentioned before. They also include DSPs and memory blocks, even though these last ones are limited in size and special care needs to be taken at design time. The interconnection among these blocks is also programmable, which is the main reason for the flexibility of FPGAs mentioned before. Lastly, their programmable connections to external components make them very versatile for systems with many sensors and actuators, whether existing or new ones that may be required.

As mentioned in Section 1.2, the Zynq® model from Xilinx® is used in this dissertation, particularly the Zynq®-7000 and Zynq® UltraScale+™ families. They include dual-core or single-core ARM® Cortex™-A9, and 64-bit quad-core or dual-core ARM® Cortex®-A53 and dual-core ARM Cortex-R5F based PS, respectively. Both families include *in the single device* Xilinx®'s PL. The PL allows users to design their own digital circuits using an HDL and then program them onto the FPGA. This allows the creation of customized hardware solutions for a wide range of applications, from signal processing to data center acceleration. As a

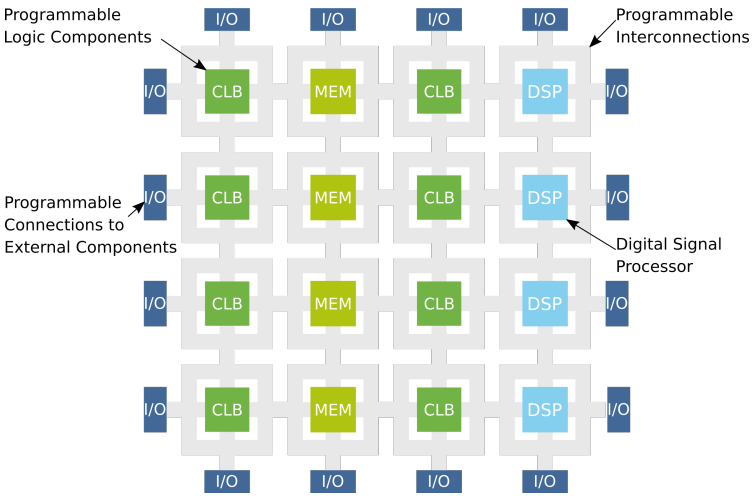


Figure 2.2: Reconfigurable computing system's diagram.

reference, the architectural overview of the Zynq®-7000 System-on-Chip (SoC) is shown in Figure 2.3.

Zynq® devices provide a good balance between performance and power consumption, making them suitable for a wide range of applications, including embedded vision, industrial control, and Internet of Things (IoT). On the other hand, UltraScale+™ is a family of FPGA devices that offers high performance and scalability for applications that require high-bandwidth data processing. These devices have a more advanced architecture than Zynq® devices, with features like high-speed serial transceivers, 28Gbps transceivers, and advanced memory interfaces. UltraScale+™ devices are typically used in high-performance computing, wired and wireless communications, and video processing applications.

For both families, the PS has a wide range of peripheral interfaces such as USB, Ethernet, UART, SPI, and I2C. These interfaces can be used to connect to various devices, such as storage devices, network devices, and sensors. Furthermore, a boot loader can be loaded on the device to initialize the hardware as well as load a Linux kernel into memory. Having a Linux kernel loaded into memory allows the OS to take control of the device and run a native Linux distribution as in a regular desktop PC with the advantages of this particular embedded system (e.g., high-performance, low-power processing).

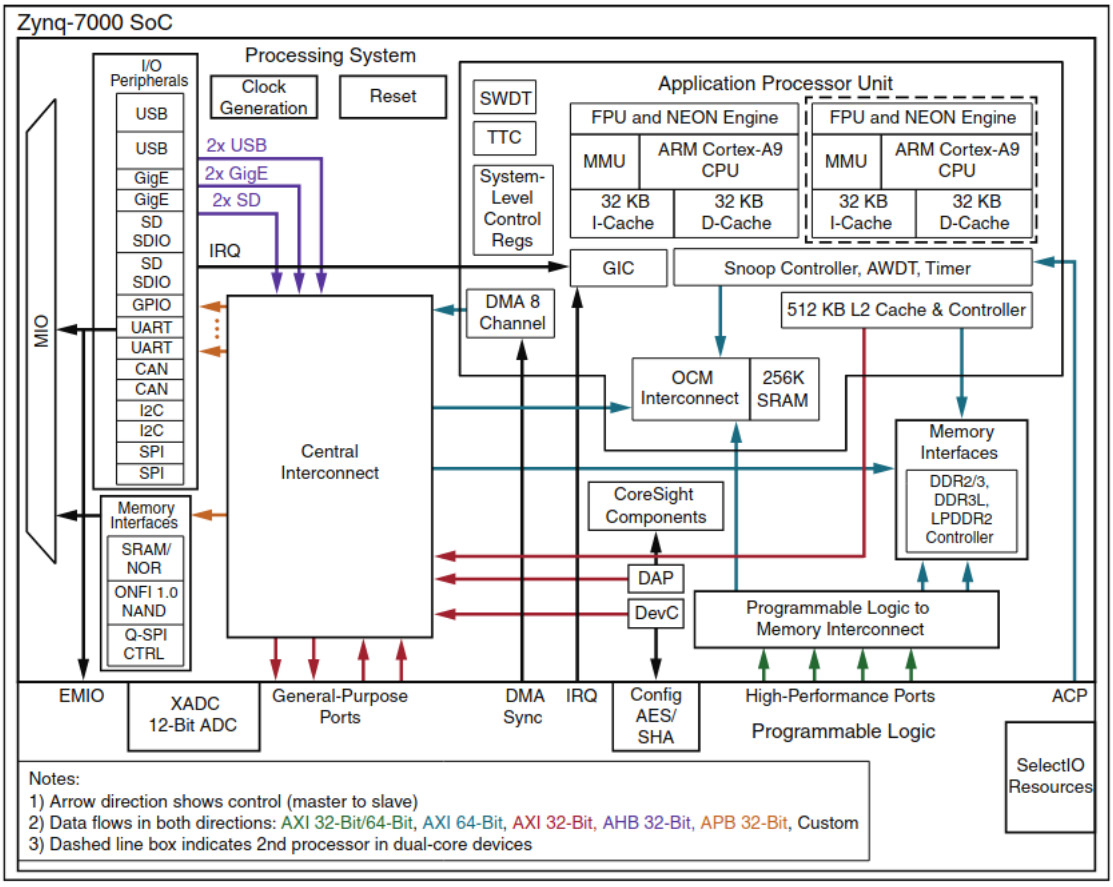


Figure 2.3: Zynq®-7000 SoC architectural overview[1]

2.2 AXI Stream Protocol

The AXIS Protocol is a widely-used interface protocol for efficient and reliable communication between digital circuits. The protocol is designed to provide a high-speed, point-to-point communication link between system components, such as between the CPU and peripheral devices like accelerators. AXIS is a packet-based protocol that operates over a single, unidirectional data channel, unlike the AXI4 protocol that supports bidirectional communication.

The protocol works in a master-slave configuration. The first one is the one that produces the data, and the second one consumes it. Four signals must always be part of the interface, namely:

- `TDATA` is the primary payload used to provide the data. The width of the data is an integer number of bits, typically 8, 16, 32, or 64.
- `TVALID` indicates that the master has data available to be transferred.
- `TREADY` indicates that the slave can consume the data produced by the master.
- `TLAST` indicates the last element to transfer (packet boundary).

For a master, `TDATA`, `TVALID`, and `TLAST` are output signals, and `TREADY` is input. For a slave, `TDATA`, `TVALID`, and `TLAST` are input signals, and `TREADY` is output.

A *frame* consists of a group of packets (bytes that are transported together across an AXIS interface), and there is a *handshake process* for the transmission to begin. For a transfer to occur, both `TVALID` and `TREADY` signals must be asserted irrespectively of their order (or at the same clock cycle). However, there are some restrictions. A master cannot wait until `TREADY` is asserted. It must always assert `TVALID` when new data is available, independently of `TREADY`. Once `TVALID` has been asserted, it must remain asserted until the handshake occurs. Similarly, a slave cannot wait until `TVALID` is asserted. It must always assert `TREADY` whenever it can consume data from the master. The slave must keep `TREADY` asserted until the handshake occurs. Figure 2.4 shows an example of a proper transmission that only starts once `TVALID` and `TREADY` have been *both* asserted, where the payload is the sequence of bytes on `TDATA`.

2.3 Model-Driven Engineering

MDE is a software engineering approach that focuses on creating models that capture a system's structure, behavior, and functionality and using those models to generate code and other artifacts. The goal of MDE is to improve the efficiency and quality of software development by emphasizing using models as a primary artifact throughout the development process. MDE is often used in complex systems, where the use of models can help manage the system's complexity and improve the development team's productivity. It can also be helpful in domains with strict requirements, as using models can help ensure that the system meets those requirements.

[1] Image taken from `https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview`

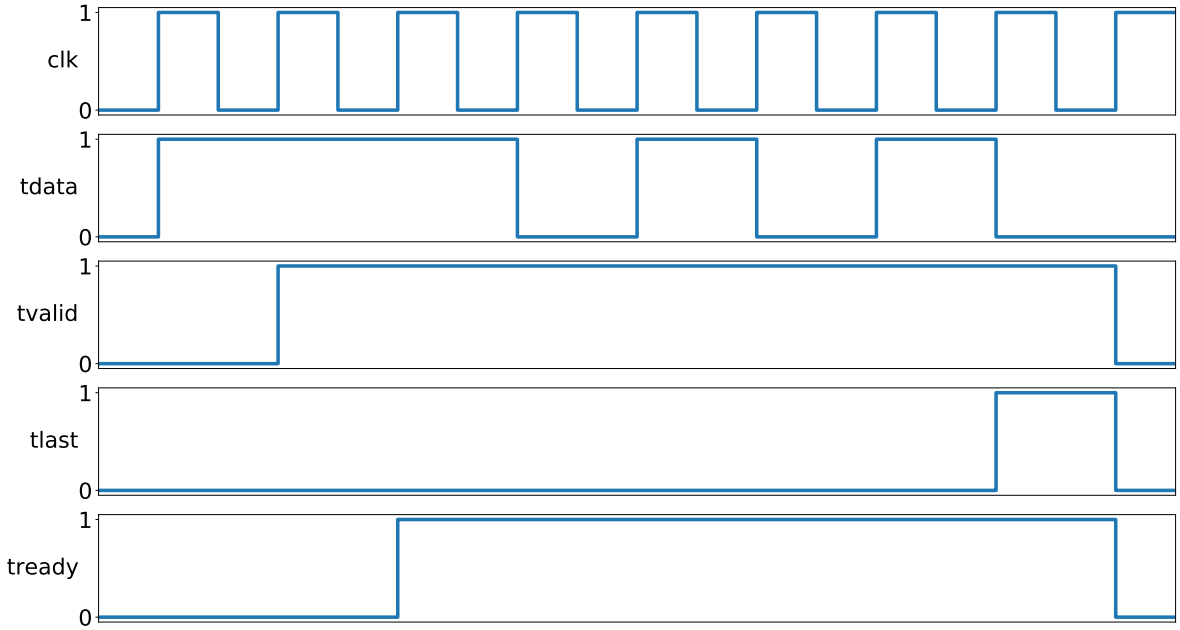


Figure 2.4: AXI Stream protocol example

MDE is designed to improve software development productivity by boosting compatibility between systems, simplifying the design process, and easing communication among individuals and teams working on the system. One key advantage of MDE is that it facilitates the reuse of standardized models, which can help simplify the development process and promote compatibility between different systems. In addition, MDE promotes the use of models to capture recurring design patterns in the application domain, which can simplify the design process and reduce the likelihood of errors.

Another important benefit of MDE, vital for this dissertation, is that it promotes communication between individuals and teams working on the different parts of a system, where experts in different domains work. By standardizing the terminology and best practices used in the application domain, MDE can help ensure that everyone involved refers to the same concepts (from their expert knowledge), which can reduce misunderstandings and improve the overall quality of the system.

One common technique used in MDE is model-driven development, in which technical artifacts such as source code, documentation, and tests are generated algorithmically from a domain model [32]. This approach can help reduce the amount of manual effort required during the development process and can also improve the accuracy and consistency of the resulting artifacts.

The key MDE aspects used along this dissertation are:

- ***Modeling***: creating models representing the system being developed or generated. These models may include high-level models of the system's architecture, as well as more detailed models of individual components and their interactions.
- ***Transformation***: using automated tools to transform the models into other artifacts, such as source code, test cases, and potential documentation.

- *Validation*: verifying that the models and their transformations are correct and consistent with the system's requirements.
- *Simulation*: using the models to simulate the system's behavior under different conditions, such as different inputs or system configurations.

2.4 The Building Blocks of Languages in Computer Science

In order to understand the main concepts exploited in Chapters 5 and 6, some language-related definitions used in computer science are required, which are explained below:

Grammar: are a set of rules that define the structure and syntax of a language, including the relationships between its elements.

Context-Free Grammar (CFG): is a formal grammar whose production rules can be applied to non-terminal symbols regardless of its context, that can generate a language. It is a formalism used to specify the syntax of a language in a way that is independent of any particular implementation. In particular, in a CFG, each production rule is of the form $A \rightarrow \alpha$, with A a single non-terminal symbol, and α a string of terminals and/or non-terminals.

Backus-Naur Form (BNF): is a type of meta-language used to formally describe the syntax of programming languages and other computer languages. It is a notation of CFG that it is used to define the set of valid strings in the language. Variables (non-terminals) are enclosed via special brackets "<var>" to distinguish them from terminal symbols. The symbol "::=" is used to indicate an equivalence similar to the derivation function in CFGs ($\rightarrow$). The symbol "|" is used to separate alternatives. Listing 2.1 shows the grammar that can be used to process expressions like $3 * (4 + 2) + 8$. This is also used to show further definitions.

Semantics: refers to the meaning of language and symbols in a particular context. For a programming language, semantics describe what a program does and what values it produces rather than how it does it. They can specify the rules for evaluating expressions, defining variables, and executing statements or how a program should behave in different situations.

Abstract Syntax Tree (AST): is a data structure used in compilers and interpreters to represent the structure of a program. It is a tree-like representation of the source code or a model, where each node in the tree represents a construct in the element (i.e., language, specification) being parsed.

Constructs: can be anything from a simple variable declaration to a complex function definition. The construct is an abstraction of the source code that captures the meaning of a particular code structure. For example, a function definition is a construct in many programming languages. The construct represents the entire function, including its name, arguments, and body. The construct provides a high-level view of the function, allowing the parser to understand the relationship between the different elements in the source code.

Token: is a sequence of characters in the source code that represents a single unit of meaning. The lexer or lexical analyzer generates tokens by breaking down the source code. Tokens can be keywords, identifiers, operators, and literals. Usually, the lexer reads the source code character by character and groups them into token based on predefined rules. The tokens are passed on to the parser, which processes them and generates a parse tree, which represents

```
1 <expr> ::= <term> "+" <term> //Non-terminal
2 <term> ::= <factor> "*" <factor> | <number> //Non-terminal
3 <factor> ::= "(" <expr> ")" | <number> //Non-terminal
4 <number> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" //Terminal
```

Listing 2.1: Example of a BNF grammar

the structure of the code based on the grammar rules. Tokens can be either terminals or non-terminals. An example is shown in Figure 2.5.

Terminal: also known as a leaf node, is a basic symbol in a grammar that represents the smallest possible unit of the language being parsed. Terminals are usually represented as keywords, identifiers, operators, or other literal values in the source code (c.f.,Lines 1 to 3 in Listing 2.1).

Non-Terminal is a symbol in a grammar that represents a higher-level structure in the language being parsed. Non-terminals are usually represented as variables or expressions that are composed of other symbols, including terminals and other non-terminals (c.f.,Line 4 in Listing 2.1).

Production rules: are a fundamental concept in formal language theory and are used to define the syntax of a language. They specify how the components of a language, such as terminals and non-terminals, can be combined to form valid sentences or expressions in the language. In the context of parsing and compiler construction, production rules are used to define the grammar of a language, which is then used by a parser to recognize and analyze the syntax of a program written in that language. Production rules consist of two parts: a Left-Hand-Side (LHS), which specifies a non-terminal symbol, and a Right-Hand-Side (RHS), which specifies a sequence of terminals and/or non-terminals that can replace the non-terminal symbol on the LHS. The LHS and RHS of a production rule together define a single transformation of a language, where the non-terminal symbol on the LHS is transformed into the sequence of symbols on the RHS. A language's set of all production rules defines its syntax, and a parser uses these rules to analyze and interpret programs written in the language. An example is shown in Listing 2.1. The characters ::= divide the LHS and RHS of the production rules.

Attributes: are values that describe properties of elements in a grammar (i.e., types, values). An attribute can be a *reference* to an AST node. Therefore, attributes can connect different AST nodes to each other (forming a graph). They can be synthesized or inherited:

- Synthesized Attributes: All the information is available on the values of other attributes in the same node or its children (subtree) (Figure 2.6a).

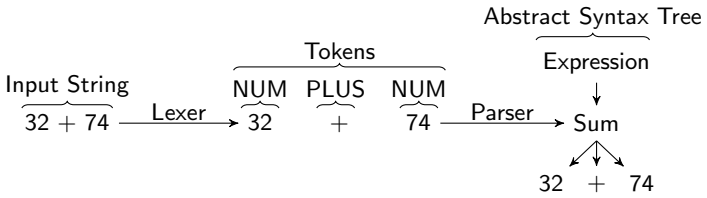


Figure 2.5: Example of how a lexer and a parser generate an AST

- Inherited Attributes: All the information is available outside the subtree (parent) (Figure 2.6b).

Attribute Grammars: used to specify the syntax and the semantics of programming languages or a Domain Specific Language (DSL). They are an extension of traditional grammars by associating additional information (attributes) with elements of the grammar, making it possible to generate code automatically.

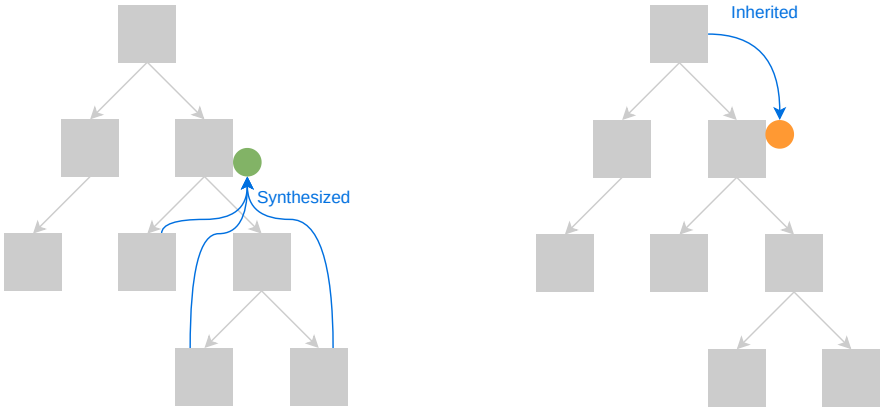
Reference Attribute Grammar (RAG): is an attribute in an abstract syntax tree that holds a reference to another node in the tree. The value of a reference attribute is determined by the node it references. Reference attributes help create relationships between nodes in the tree, such as parent-child relationships or relationships between siblings. There is no need in RAGs to replicate the information available in the syntax tree into attributes as the AST can be used as the information source by using RAGs [33].

A lexical analyzer, also known as a lexer, transforms an input stream of characters into a stream of tokens, which are the smallest units that a parser can handle. A syntactic analyzer, also known as a parser, converts the input stream of tokens into an attributed syntax tree [34]. Thereby, the AST is generated by the parser, which processes the tokens generated by the lexer and uses the grammar rules to construct a tree-like structure that represents the program's structure (Figure 2.5). The AST captures the program's logical structure, including its syntax, control flow, and relationships between different elements in the source code. In this context, each node in the tree represents a single construct in the language being parsed. The nodes are connected to form a tree-like structure, where the parent node represents a higher-level construct, and the child nodes represent lower-level constructs. This tree-like structure provides a hierarchical representation of the source code that captures the relationships between different elements in the program.

Once the AST is generated, it can be used for various purposes, such as code optimization, code analysis, code generation, and others. Compilers and interpreters also use the AST to generate machine code or execute the program. In summary, the AST is a crucial component of the language processing pipeline, providing a structured representation of the program that can be used for various purposes.

Considering the BNF grammar example from Listing 2.1 that takes the mentioned input $3 * (4 + 2) + 8$, the lexer will generate the following list of tokens: <*number:3*>, <*symbol:**>, <*symbol:(*>, <*number:4*>, <*symbol:+*>, <*number:2*>, <*symbol:)*>, <*symbol:+*>, <*number:8*>. These will be taken by the parser to construct the syntax tree shown in Figure 2.7.

The following section describes the compiler framework used in this dissertation.



(a) Synthesized Attribute (b) Inherited Attribute

Figure 2.6: Synthesized vs. inherited attributes

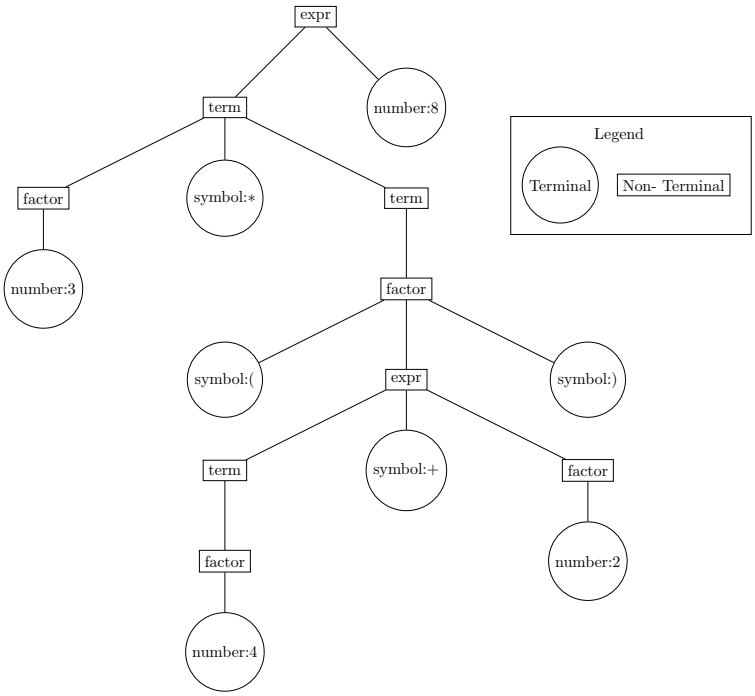


Figure 2.7: Generated syntax tree for BNF grammar example for input string $3 * (4 + 2) + 8$

2.5 JastAdd: The Meta-Compilation System

JastAdd [35, 36] is a flexible system that allows compiler behavior to be implemented conveniently based on an object-oriented AST [35]. The "meta" in "meta-compilation" refers to the fact that *JastAdd* can be used to generate compilers for other programming languages instead of being a compiler on its own. Then, being a "meta-compiler" makes it possible to use *JastAdd* to develop custom compilers that can parse, analyze, and generate code for a specific programming language or DSL. The compilers built with *JastAdd* are tailored to the needs of the particular language or domain, making it possible to add new features and functionality in a flexible and modular manner.

JastAdd is a compiler framework that facilitates the creation of compilers, parsers, and interpreters for programming languages [37]. The name *JastAdd* implies the ease of extensibility: just add to the AST. These are the three main ideas contribute to the modularity and extensibility of *JastAdd*:

1. Attribute Grammars are a type of grammar that describe a tree-based data structure, where each node in the tree has a set of attributes. These grammars can be used to define the semantics of a programming language, and *JastAdd* provides support for writing attribute grammar specifications.
2. *JastAdd* is also object-oriented, as it is implemented in Java and provides a set of classes and methods for building compilers and extensions. This object-oriented design makes it easy to add new functionality to the framework and reuse existing code when building new compilers. The object-oriented architecture also allows for the integration of *JastAdd*-based compilers with other tools and libraries, such as debuggers and performance profiling tools.
3. *JastAdd* supports declarative thinking through the use of attribute grammars. Attribute grammars allow specifying the properties and relationships in a declarative manner. This means it is possible to describe *what* the language should do rather than how it should be done, making implementing compilers and extensions much simpler. By specifying the language in a declarative way, *JastAdd* provides a higher level of abstraction for writing compilers, making it easier to understand and maintain the code. This approach can also make it easier to reason about the behavior of the language, as the focus is on the desired properties and relationships rather than the implementation details. Additionally, the declarative specification can be more concise and easier to read than imperative code, making it easier to collaborate and share knowledge about the language.

These three modularity and extensibility points make *JastAdd* the proper framework to fulfill the aims of this dissertation.

It is built with a Java-based architecture that offers flexibility and modularity for designing and implementing language extensions and compilers. Attributes can be seen as methods of AST nodes. They are similar to abstract methods[2], and equations are similar to method implementations. The framework enables the development of custom programming languages, the creation of DSLs, or the augmentation of existing programming languages. The key feature of *JastAdd* is that it allows properties (attributes) of AST nodes to be programmed declaratively.

[2]Template for methods in related subclasses providing a common interface among them.

The attributes can be simple values like integers, sets, or reference values pointing to other nodes in the AST [37]. This is a crucial aspect, as it enables the explicit definition of the graph properties of a program. Therefore, as nodes in the AST are objects with including attributes, the resulting data structure is an *object-oriented graph model* rather than a simple AST.

JastAdd's ability to declaratively program the attributes of abstract syntax tree nodes is a crucial feature. This results in an object-oriented graph model, implemented using Java classes, with the attributes forming a method Application Programming Interface (API) for those classes. Attributes may have parameters, making it possible to transfer or assign complex computations from one node to another by accessing a parameterized attribute through a reference attribute.

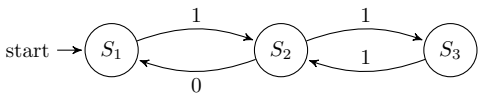
The parser builds the abstract syntax tree and calculates the values of attributes using attribute grammars, which are defined using equations. The method API can then be used to access these values. There are two noteworthy points to consider. First, it does not matter the order in which the attributes are evaluated. Second, it is possible to add new attributes, equations, and syntax rules, which facilitates the extensibility of the language or DSL.

Table 2.1 showcases the syntax specification for the AST used in *JastAdd*. An example of the grammar for a language that generates state machines can be seen in Listing 2.2. The root of the AST is `StateMachine` with a *list* of `Declaration` components. The purpose of the abstract declaration is to provide a common base type for all specific declarations. This allows them to be processed in a generic manner. In this grammar, a `State` and a `Transition` are both considered to be `Declaration`. However, they have different properties. The specific properties of `State` and `Transition` can be captured by making it abstract. At the same time, they can still be processed in a common way. This makes writing code that can handle all types of declarations easier without repeating the same logic for each specific type of declaration. `State` contains only the `Name` component denoting the name of the state. `Transition` has three components, namely `Source`, `Target`, and `Condition`. These components express from which state to which state the transition should happen and the condition to trigger it. Figure 2.8a shows a custom DSL that describes the state machine depicted in Figure 2.8b. The VHDL code, *written manually* for it, is shown in Listing 2.3.

The goal is to have a generic approach, to automatically generate VHDL code for any state machine specified via the custom DSL. For this, the defined grammar in Listing 2.2 is used to obtain an AST representing the desired state machine. The DSL is parsed, and the resulting populated AST for the example shown in Figure 2.8 is displayed in Figure 2.9. In the next section, the function of template engines is discussed, outlining the process of how this populated AST is utilized to obtain the VHDL code shown in Listing 2.3, as well as any

```
1    state S1;
2    state S2;
3    state S3;
4    trans S1->S2:input=1;
5    trans S2->S1:input=0;
6    trans S2->S3:input=1;
7    trans S3->S2:input=1;
```

(a) Example of a DSL for state machines



(b) Diagram of the expected state machine

Figure 2.8: Custom DSL and graphical representation of a desired state machine

Table 2.1: AST specification syntax used in JastAdd

Syntax	Meaning
`A;`	AST class
`B:S;`	AST class, abstract
`B::=Y;`	Child component Y
`B::=MyY:Y;`	Child component MyY of type Y
`X::=C*;`	List component C, containing C nodes
`X::=MyC:C*;`	List component MyC, containing C nodes
`Y::=[D];`	Optional component D
`Y::=[MyD:D];`	Optional component MyD of type D
`Z::=<E>;`	Token component E of type String
`Z::=<F:Integer>;`	Token component F of type Integer
`U::=/V/;`	NTA component V
`U::=/G:V/;`	NTA component G of type V

```
1 StateMachine ::= Declaration*;
2 abstract Declaration;
3 State : Declaration ::= <Name:String>;
4 Transition : Declaration ::= <Source:String> <Target:String> <Condition:String>;
```

Listing 2.2: *JastAdd* grammar for state machines

other state machine specified with the custom DSL. The complete process is depicted in Figure 2.10.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity state_machine is
5      Port ( clk : in  STD_LOGIC;
6             input  : in  STD_LOGIC;
7             current_state : out  STD_LOGIC_VECTOR (1 downto 0));
8  end state_machine;
9
10 architecture Behavioral of state_machine is
11     type state_type is (S1, S2, S3);
12     signal present_state, next_state : state_type;
13 begin
14     process (clk)
15     begin
16         if (clk'event and clk = '1') then
17             present_state <= next_state;
18         end if;
19     end process;
20
21     next_state_process : process (present_state, input)
22     begin
23         case present_state is
24             when S1 =>
25                 if (input = '1') then
26                     next_state <= S2;
27                 end if;
28             when S2 =>
29                 if (input = '1') then
30                     next_state <= S3;
31                 end if;
32                 if (input = '0') then
33                     next_state <= S1;
34                 end if;
35             when S3 =>
36                 if (input = '1') then
37                     next_state <= S2;
38                 end if;
39                 if (input = '0') then
40                     next_state <= S3;
41                 end if;
42         end case;
43     end process;
44
45     current_state <=
46         "00" when present_state = S1 else
47         "01" when present_state = S2 else
48         "10" when present_state = S3;
49 end Behavioral;
```

Listing 2.3: Manually coded state machines in VHDL

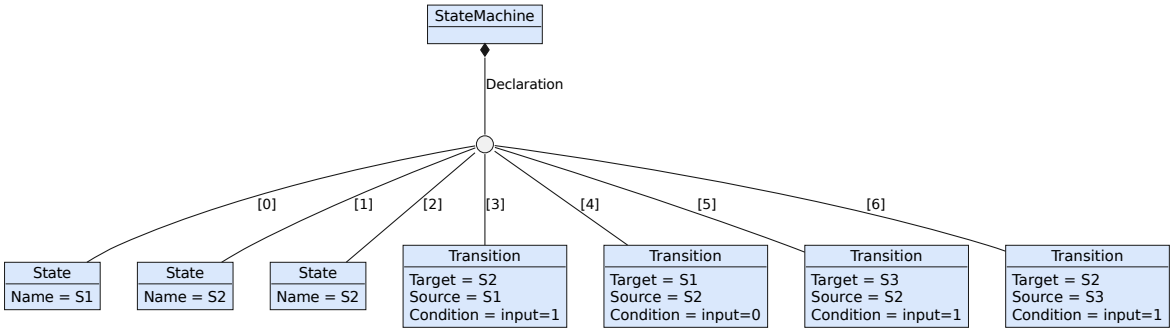


Figure 2.9: Populated AST for the example state machine

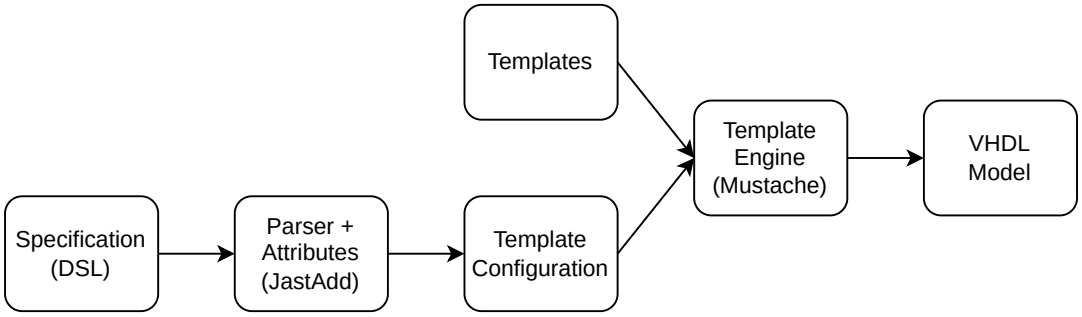


Figure 2.10: Code generation process used in this dissertation

2.6 Template Engines

A template engine is a software component that is used to generate source code from a set of templates and data sources (i.e., template configuration). The templates consist of placeholders and control structures, such as conditional statements and loops, that define the structure and content of the output. The data sources provide the values that are used to populate the placeholders in the templates. The advantage of using a template engine is that it separates the presentation of the data from its underlying logic and data storage.

Templates provide a way to separate the presentation of information from the underlying data and logic. In this way, templates allow the implementation of a clean separation of concerns, where the data and logic are managed in one part of the code, and the presentation of that data is handled in another part. This makes it easier to maintain and update the code, as changes to the outcome (generated code) can be made independently of changes to the data and logic. Additionally, templates can also help to encapsulate the implementation details and hide them from the generated code, promoting modularity and encapsulation.

Mustache [3] is the logic-less template engine used in this dissertation. The templates themselves only specify what should be displayed and not how it should be displayed (hence logic-less). This separation of concerns allows developers to work on the logic of the application and the presentation of the data independently, making it easier to maintain and update the code. Mustache templates are written in a simple syntax, using placeholders to denote the insertion of data. This data can be passed to the template engine as a context, containing the actual values inserted into the template. With the use of attributes, the populated AST

[3] Mustache—Logic-Less Templates, `https://mustache.github.io`

can be traversed to derive the necessary information and obtain the template configuration shown in Listing 2.4. The combination of said template configuration with a template shown in Listing 2.5 allows the template engine to produce the expected code, also known as an artifact.

The mustache syntax works as follows. Everything that is between {{}} is a place holder, such as Line 5 in Listing 2.5 where the content of `entity_name` from the template configuration (Line 1) will be replaced in the artifact. {{*#NAME*}} acts as a conditional *True* statement whereas {{`^NAME`}} is its *False* counterpart, such as Line 38 or Line 41 in Listing 2.5. Both statements are closed with {{*/NAME*}}, as shown in Line 40 or Line 43. Everything that is located in a conditional statement will be part of the artifact, depending on whether the condition is met. One last thing that is part of the mustache syntax are *partials* templates, which are not shown in this example but are later used in the following chapters. They are reusable templates that can be included in a main template, which contain a specific portion of the overall template that is used multiple times throughout the main template. They can be considered as recursive templates, helpful in expanding lists of lists, for example.

From the next section onwards, the state-of-the-art in acceleration focused on robotics is presented, software aspects related to middlewares, and MDE works to generate robotics systems easily.

```
1  entity_name: state_machine
2  state_type: (S1, S2, S3)
3  states:
4    - state_name: S1
5      transitions:
6        - input: 1
7          target_state: S2
8    - state_name: S2
9      transitions:
10       - input: 1
11         target_state: S3
12       - input: 0
13         target_state: S1
14   - state_name: S3
15     transitions:
16       - input: 1
17         target_state: S2
18       - input: 0
19         target_state: S3
20 output:
21   - value: "00"
22     present_state: S1
23   - value: "01"
24     present_state: S2
25   - value: "10"
26     present_state: S3
27     last: True
```

Listing 2.4: Template configuration for state machines

```
{{#state_type}}
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity {{entity_name}} is
    Port ( clk : in  STD_LOGIC;
           input  : in  STD_LOGIC;
           current_state : out  STD_LOGIC_VECTOR (1 downto 0));
end {{entity_name}};

architecture Behavioral of {{entity_name}} is
    type state_type is {{state_type}};
    signal present_state, next_state : state_type;
begin
    process (clk)
    begin
        if (clk'event and clk = '1') then
            present_state <= next_state;
        end if;
    end process;

    next_state_process : process (present_state, input)
    begin
        case present_state is
            {{#states}}
            when {{state_name}} =>
            {{#transitions}}
                if (input = '{{input}}') then
                    next_state <= {{target_state}};
                end if;
                {{/transitions}}
            {{/states}}
        end case;
    end process;

    current_state <=
        {{#output}}
        {{#last}}
        "{{value}}" when present_state = {{present_state}};
        {{/last}}
        {{^last}}
        "{{value}}" when present_state = {{present_state}} else
        {{/last}}
        {{/output}}
end Behavioral;
{{/state_type}}
```

Listing 2.5: Mustache template for state machines

2.7 Robotic Applications in Adaptive Computing

Even though the main focus of this work is adaptive computing (i.e., FPGAs), GPUs cannot be left out for a comprehensive literature review. This is due two two reasons. The first one is the competitiveness in terms of programmability and the second one is availability, specially in the last years. They both have different advantages and disadvantages, depending on the applications, which are shown below. Different robotic applications have been proposed, targeting FPGAs and GPUs as their main PE. It is important to start this study with the different applications in robotics, which are increasing their complexities due to a large amount of data that needs to be processed from several sensors and how accelerators (i.e., FPGAs and GPUs) are needed to ease this. This section explores them to understand which type of applications benefit of these alternative to accelerate compute-intensive tasks.

2.7.1 FPGA Applications

Robotic systems are generally complex ones as they integrate different technologies and multiple heterogeneous sensors and actuators. This means that many algorithms are needed to obtain meaningful information from raw sensor data to perform specific actions via actuators, imposing several challenges for designers. On the one hand, there is a limitation concerning the amount of onboard resources, such as memory storage and computational power, making it hard to meet real-time constraints. On the other hand, most state-of-the-art robotic systems will have power constraints, so efficient designs are needed to cope with this.

FPGAs have been attracting attention in the research community as an energy-efficient PE. Their specialized design hardware logic allows FPGA-based accelerators to surpass CPUs and GPUs in terms of performance and energy efficiency [38]. Their main advantage is the possibility to achieve high energy efficiency via *custom hardware designs* compared to CPUs or GPUs. However, it is pretty challenging due to the more significant design effort required. The span of robotic applications proposed in FPGAs ranges from sensing (extracting meaningful information from raw sensors data), perception (building a representation of the robot's environment), and decision (control of actuators). Here some of them are explored, understanding which are the goals of the research community to solve the challenges mentioned before, in this case via hardware acceleration.

Much effort has focused on perception (e.g., stereo vision, object detection, semantic, classification). Navigation and obstacle avoidance are two popular applications that have been increasing in popularity lately, relying on real-time and stereo vision systems. This consists of capturing images with two cameras from different points of view of the same scene. The disparities between the corresponding pixels in both images are searched with matching algorithms, and the depth information can be computed from the inverse of the disparity [38]. Stereo matching algorithms can be divided into local (compute disparity by processing and matching the pixels around the point of interest) and global (compute disparity by matching all pixels and minimizing a global cost function) algorithms. The first ones are faster and high parallelizable, but their accuracy is lower. The latter ones require more resources to achieve higher accuracy.

Perception

Jin and Maruyama [39] presented a stereo matching local algorithm using Cost Aggregation (CA) and Fast Locally Consistent (FLC) to achieve a low error rate while maintaining a high processing speed. The focus was on the RTL, and they achieved 507*fps* for 640*x*480 pixel images. The evaluation was done on a Xilinx Virtex-6 and compared with two GPUs (GTX480 and 7900GTX). Results show a lower FPGA power consumption by one order of magnitude and one order of magnitude higher for the processing speed. The authors concluded that the resource utilization of their proposed algorithm is well suited for modest-size FPGAs.

Wang et al. [40] focused on Semi-Global Matching (SGM), which computes disparity by comparing local pixels, and then approximates an image-wide smoothness constraint with global optimization. Despite more robust disparity maps, the challenge for this approach is the need for more storage resources. The implementation also focused on the RTL design and was evaluated on an Altera Stratix-IV. They achieved 67*fps* and 42*fps* for 1024*x*768 and 1600*x*1200 pixel-images respectively.

While both previous works implemented at RTL, Rahnama et al. [41] used HLS for their SGM variation. Their evaluation was a Xilinx ZC706 FPGA, achieving similar levels of accuracy compared to related work while reducing the power consumption by two orders of magnitude. In their case, 72*fps* were achieved for 1242*x*375 pixel-images. Overall, for image processing and stereo vision applications, these publications [39, 40, 41] and others alike showed that FPGA-based designs obtained higher energy efficiency compared to GPUs and CPUs. However, higher effort in terms of design and fine-tuning was required.

As far as space applications, Malin et al. [42] reports on the new Rover that was sent to Mars. It heavily relies on FPGAs for scientific instrument control, image processing, and communications. Specifically for image processing, it consists of a Xilinx Virtex-II with a Microblaze soft-core processor. All core functionalities (timing, interface, and compression) are implemented as RTL which are peripherals for the Microblaze.

Navigation and Obstacle Avoidance

Another highly parallelizable application is Simultaneous Localization and Mapping (SLAM), used for path planning, which has had much attention as well. Gautier et al. [43] presented the implementation of two low-power 3D reconstruction algorithms, namely “Iterative Closest Point” and “Volumetric Integration”. The work was based on an Altera Stratix V FPGA using OpenCL by porting the original CUDA implementation of each algorithm. This allowed them to do some optimizations for memory access and synchronization between computing units. Moreover, they improved the original code manually using a double nested loop with one index being a function of the other, resulting in a poor optimization for the compiler. The solution was to remove the index dependencies and unroll the loops manually.

Abouzahir et al. [44] evaluated the processing time of different SLAM algorithms in several embedded systems. They concluded that Fast-SLAM2.0 is the ideal compromise between accuracy and computational time. Then, the algorithm was optimized to implement it on GPU with combining OpenCL and OpenGL and on FPGA with HLS. The FPGA implementation obtain a 7.5x acceleration with respect to the GPU one.

Boikos and Bouganis [45] focused on the acceleration of Large-Scale Direct Monocular (LSD) SLAM in a SoC Zynq-7020. In their case, the intermediate data produced is too large, so it is shared between accelerators via DDR memory. They highlighted the importance of optimizing memory architectures (e.g., data movement, caching) in an application like this to ensure the scalability and compatibility of the design.

Murray et al. [46] presented an FPGA-based architecture for motion planning, focusing on collision detection. In this case, compared to previous work [45], they avoid storing and accessing pre-computed data in memory. Instead, data is encoded as a binary representation and create logical circuits to represent collision data, hence the reason to call the approach "microarchitecture". They achieved sub-millisecond speed for motion planning and an improvement for power consumption of one order of magnitude with respect to a GPU implementation.

Bondhugula et al. [47] proposed a parallel FPGA-based implementation for graph search, which is the next thing to compute after collision detection. This algorithm tries to find the shortest and safe path to the targeted position. In this case, the authors achieve a 15x speedup compared to an optimized CPU-based implementation.

A first conclusion to draw from the summary shown here is that FPGA-based designs have shown a higher energy efficiency compared to GPUs and CPUs for most image processing applications [38] and other parallelizable applications. It can be depicted from Table 2.2 that the more complex the application is, the higher the abstraction of the design (HLS rather than HDL). This is usually because most modern tools take care of the data movement, leaving the designer to focus only on the targeted problem's acceleration (e.g., stereo-matching, SLAM). However, incorporating the resulting accelerators contributes to the integration problem mentioned earlier in general for robotic systems. Most of the works presented here adopt a similar FPGA-SoC architecture, accelerating the most compute-intensive parts of the algorithms. However, integrating these applications into existing robotic environments is usually unattended.

Note that Table 2.2 summarizes the different applications that have been proposed on FPGAs. It does not cover by all means each of the *applications* or *fields* mentioned there. The reader is encouraged to follow up on [38] for a more fine-grained literature review on each topic.

A vast range of applications in robotics can be parallelized and thus accelerated on FPGAs. However, there are two main challenges to face. On the one hand, extra effort is needed to integrate them into already running systems. On the other hand, the complexity of achieving energy-efficient implementations is high, despite newer techniques such as HLS.

Table 2.2: FPGA applications in robotics

Reference	Platform	Application	Focus	Abstraction Level	Metric	Result
Jin and Maruyama (2014) [39]	Xilinx Virtex-6	Stereo vision	Accelerator	RTL	FPS	507 (640x480), 199 (1024x768) 76 (1920x1080 -full HD-)
					Power	10.6W
Wang et al. (2015) [40]	Altera Stratix IV & V	Stereo vision	Accelerator	RTL	FPS	68 (1024x768), 43 (1600x1200)
Rahnama et al. (2018) [41]	Xilinx ZC706	Stereo vision	Accelerator	HLS	FPS	301 (38x4288), 198 (450x375) 109 (640x480), 72 (1242x375)
					Power	3W
Gautier et al. (2014) [43]	Altera Stratix V	SLAM	CPU-FPGA Architecture	OpenCL	FPS	28
Abouzahir et al. (2018) [44]	Altera Arria 10	SLAM	Accelerator	HLS, OpenCL & OpenGL	FPS	102
Boikos and Bouganis (2016) [45]	Xilinx Zynq-7020	SLAM	HW/SW Co-Design	C & HLS	FPS	4.55 (320x240)
					Power	2.25W
Murray et al. (2016) [46]	Intel Xeon	Motion planning	HW/SW Co-Design	C & RTL	-	-
Bondhugula et al. (2006) [47]	Intel Xeon	Graph search	HW/SW Co-Design	C & RTL	Speedup	15x (average)
Malin et al. (2017) [42]	Xilinx Virtex-II	Space	HW/SW Co-Design	C & RTL	FPS	6 (1280x720)

2.7.2 GPU Applications

GPUs are specialized circuits designed to rapidly manipulate and vary memory to accelerate the creation of images in frame buffers, most commonly used in embedded systems, mobile phones, personal computers, workstations, and gaming consoles. They achieve higher efficiency than general-purpose CPUs due to their highly parallel structure [48]. Even though multiple works have compared GPUs and FPGAs [49, 50, 18], they mainly based the comparison on workstations or personal computers.

The main interest here is robotic applications, so the works presented in this section are mainly related to that field. This mainly refers to embedded systems, targeting a balance between accuracy, throughput, and power budget. These objectives are crucial for applications in several domains such as robotics, autonomous driving, and drones [51]. Therefore, the focus is on *embedded GPUs*, mainly NVIDIA's Jetson, being one of the most widely used ones as it provides high performance per watt due to its performance-efficient and low-power GPU cores. It is essential to highlight that most works over this platform focus on deep learning models. They can be split into those that have a pure GPU implementation and those that combine GPUs with CPU.

Computation on GPU

Most of these works rely mostly on the GPUs to do the computation and use the CPUs mainly as data movers. As far as targeting embedded systems, Hegde and Kapre [52] present a Caffe[4]-compatible tool for generating and optimizing code, targeting devices with a power budget of up to 20W. The evaluation is based on a comparison among a GPU (Jetson TX1), DSP (TI Keystone II), RISC+Network-on-Chip (NoC)-based multicore (Parallella's Epiphany-V), and an FPGA (Xilinx ZC706). One main difference is that the DSP and FPGA perform pixel operation in 16 bits fixed-point format, whereas GPU and Epiphany-V support single-precision floating-point format. As far as performance and energy efficiency, the Jetson outperformed all other embedded platforms. In terms of programmability, GPUs also showed easier ways, followed by DSPs. However, to further improve the performance by optimizing the designs, Epiphany-V and especially FPGAs provide better outcomes at the expenses that they require more effort to ensure a correct operation and better results in terms of performance improvements.

Pierre [53] focused on perception and visuomotor control to allow a robot to follow another one. This work presented a technique that uses a spatio-temporal Deep Neural Network (DNN) with only RGB images from a camera as an input. It perceives the robot's motion by studying its environment and relative velocities to other objects. Interestingly, due to the memory capacity of DNNs, this technique allows the leader robot to be out of sight for short periods.

Wang et al. [54] proposed a lane detection algorithm for autonomous driving. Their algorithm is split into two steps. The first one classifies each pixel, whether it belongs to a lane, to estimate a lane edge. The second step localizes the lanes based on the estimations. They achieved remarkable Frames per Second (FPS), being 330 and 1300 respectively for the two steps involved in the algorithm on a Titan XP GPU. The overall performance reaches 250 FPS. 26 FPS are achieved on a Jetson TX1.

[4] Deep learning framework (`https://caffe.berkeleyvision.org/`)

Regarding drone navigation, Sanket et al. [55] presented an approach for a quadcopter to fly through a gap by only using a monocular camera and onboard sensors. The technique is based on finding the contour of an opening as the position where the discrepancy in spatial depth is maximum. The work is based on FlowNet[5] and a PID controller for the altitude and position of the drone. In this case, the chosen platform is a Jetson TX2 running both the vision and control algorithms. Another drone application is presented by Madaan et al. [56] to detect wires by only using a monocular camera for perception. Their work is based on a Jetson TX2, achieving up to 4.4. FPS with higher precision and speed than previous techniques.

Attaran et al. [57] presented a personal monitoring system based on two machine learning classifiers, including Support Vector Machine (SVM and k-nearest neighbors (KNN)). They used four different physiological sensors, requiring multiple sampling and processing capabilities with low-power consumption requirements. Hence, they proposed a reconfigurable processor for SVM and KNN for personalized stress detection. In their case, a comparison among an embedded CPU (ARM A53) (used as a baseline), GPU (Jetson TX1 and TX2), and FPGA showed an improvement in the energy efficiency of the latter one by two orders of magnitude compared to the GPUs. An evaluation with (post-layout) ASIC was performed, showing the best results. However, this option has the drawback of high costs and a longer time to market. The authors concluded that even though GPUs still offer better energy efficiency than the baseline, FPGAs would be the best solution considering the high energy efficiency and accuracy besides being reprogrammable. Several other works [58, 59] followed a similar approach targeting different applications, and combine GPUs and CPUs. They all reached similar conclusions as [57], meaning that there is an active part of the research community focusing on low-power embedded systems that could be used for different applications.

A field that is increasingly showing interest in robotics is heterogeneous platforms, where several works have been proposed, relying on DNNs on GPUs and are described below.

GPU combined with CPU

Most of these works distribute the computation between CPUs *and* GPUs. Rallapalli et al. [60] investigated the feasibility of running DNNs on embedded devices. As these devices have limited memory capacity and mainly do not include memory management schemes, they concluded that large algorithms like the famous YOLO are not fitted for such devices. They evaluated several techniques for efficient memory usage, such as targeting only inference and not allocating memory for variables that are not required during this stage. They obtained a reduction from 4.4*GB* to 2.8*GB*. Besides, they split the algorithm into GPUs and CPUs in a pipelined architecture manner by offloading some operations to the CPU which includes memory management.

Otterness et al. [61] evaluated the consequences of different memory management techniques. There are three schemes in the Jetson, namely "conventional", "zero-copy" and "unified memory". The first one refers to explicitly copying data from CPU to GPU, bringing large data transfer overheads. The second one allows the CPU and GPU to access the same memory region, without the need to allocate GPU memory, but without caching. The last one is similar to "zero-copy" by sharing memory points, and the benefit is that caching is allowed. One of the main conclusions that they drew was the importance of the proper choice of CUDA to obtain the best performances for each scheme.

[5] Evolution of Optical Flow Estimation with Deep Networks

Manderson et al. [62] presented controller for the swimming robot "AQUA". The main focus is a control algorithm to guide the robot underwater to navigate close to coral reefs with obstacle avoidance. They also present a heterogeneous design by having the control algorithms on a CPU and the neural network on a Jetson. They achieved a 10 FPS with an accuracy of 41%. A similar approach was shown by Gu et al.[63], using YOLO to detect tennis balls and then perform path planning to collect them.

Table 2.3 summarizes what has been presented previously. It can be inferred that embedded GPUs devices are valuable resources when power-budget requirements are in place. GPUs devices are usually easier to program compared to FPGAs, but there would be cases where still some partition of applications or algorithms is needed. This will usually increase the complexity of the design, moreover when considering optimizations that can be done. As their programmability is closer to CPUs, there have not been many efforts from the modeling side. However, due to current trends and possibilities, robotic systems can be composed of multiple PE and similarly to FPGAs, middlewares are helpful for system integration.

Middlewares help designers to combine multiple components, which can help to address the challenge of integration. However, most research focuses on the integration of software components. Lately, efforts on integrating FPGAs with robotics middlewares have emerged, which are discussed in Section 2.8.1.

Table 2.3: GPU applications in robotics

Reference	Platform	Application or Field	GPU + CPU	Metric	Result
Hegde and Kapre (2017) [52]	Jetson TK1, TI Keystone II, Xilinx ZC706 and Epiphany-V	Handwriting recognition (MNIST) and object detection (CIFAR-10)	✗	Throughput	35 Gops/s
Pierre (2018) [53]	Jetson TK1	Perception and visuomotor control	-	Qualitative	-
Wang et al. (2018) [54]	Titan XP GPU Jetson TK1	Lane detection	✗	FPS	250 26
Sanket et al. (2018) [55]	Jetson TX2	Drone navigation	✗	Success rate	85%
Madaan et al. (2017) [56]	Jetson TX2	Drone wire detection	✗	FPS	4.4
Attaran et al. (2018) [57]	ARM A53 (baseline) TK1 TK2 Xilinx Artix-7 ASIC	Personal monitoring system	✗	Power (W) (KNN/SVM) Classifier Throughput (dec/sec) (KNN/SVM) Classifier Energy efficiency improvement over baseline (KNN/SVM) Classifier	1.48/1.53 2.12/2.09 2.43/2.61 0.728/0.702 0.076/0.039 2/5.29 130/212 225/357 195/121/1250000 243902/2941176 1x/1x 46x/29x 69x/39x 200044x/514903x 2373712x/21586294x
Abtahi et al. (2018) [58]	Jetson TK1, ARM A53 Xilinx Zynq 7020	Signal processing (FFT) (Direct-Conv/FFT-Conv/FFT-OVA-Conv)	✓	Execution time (ms) Energy (mJ) Throughput (MB/s)	36/21/12 119/103/57 10/30/1960
Jafari et al. (2018) [59]	Jetson TK2, Xilinx Artix-7, ASIC	Multimodal data classification	✓	Latency (ms) Throughput (labels(sec)) Power (mW) Energy (mJ)	0.9/2/14.8 1185/491/67 1763/175/18.5 1.5/0.35/0.27
Rallapalli et al. (2016) [60]	Jetson TK1	Object detection with YOLO	✓	Memory usage reduction	63.63%
Otterness et al. (2017) [61]	Jetson TK1	Traffic sign recognition	✓	-	-
Manderson et al. (2018) [62]	Jetson TK2	Perception and visuomotor control	✓	FPS Accuracy	10 41%
Gu et al. (2018) [63]	Jetson TK1	Object detection and path planning	✓	-	-

2.8 Robotics Middlewares

A middleware is a computer software that provides services to software applications (e.g., communication). It could be envisaged as an abstraction layer between the OS and the application running on it. Generally speaking, middlewares are the mediator between the application front-end (i.e., client) and back-end resources (e.g., hardware device) for which the client might request data. A middleware should be customizable to different scenarios and applications. Older generations of robots were designed with one task in mind and built only for that purpose. Modern ones usually follow a modular design and implementations. They can be considered complex distributed systems with many heterogeneous hardware components (e.g., sensors, actuators) and software modules. These are jointly needed to achieve a given task, but their integration is not usually trivial. Even though modularity brings benefits from the engineering perspective, it raises some integration issues such as communication, interoperability, and configuration. Relying on a middleware helps to glue all components together, supporting concurrency-intensive operations, robustness, and modularity [64]. However, they should not increment the already challenging task of developing robotic systems. On the contrary, middlewares should simplify the development process by providing an abstraction layer with simplified interfaces. They should also provide efficient communication and simple interoperability mechanism modules. These become essential characteristics of the abstraction for the heterogeneity of hardware components to share data among them and their software counterparts. Ideally, middlewares should provide real-time interaction services with other systems considering ubiquitous robotic systems' interaction.

Multiple works and research projects have focused on the issues mentioned before, primarily from the software perspective. An early survey [64] identified the following objectives and grouped several approaches accordingly:

- Enhancing the development process by providing some form of modular design mechanism, high level of abstraction, and component-based development.
- Reusability of existing components.
- Better utilization of resources and real-time support.
- Integration with external components.

This work covers mainly the integration of robotic systems but from a hardware perspective. Therefore, only a brief overview of the most relevant existing middlewares is discussed next, as a motivation for this dissertation.

The *Orcos Project* developed a general-purpose modular framework for robot and machine control [65]. The Real-Time Toolkit (RTT) and Orcos Component Library (OCL) established a component-based infrastructure and a library of ready-to-use components, providing the high-level management of interactions within an application.

Yet Another Robot Platform (YARP) [66] aims to minimize the effort devoted to infrastructure-level software development by facilitating code reuse, modularity and so maximize research-level development and collaboration. It supports building a robot control system as a collection of programs communicating in a peer-to-peer way, with an extensible family of connection types (e.g., TCP, UDP, multicast, local, MPI) that can be swapped depending on the needs of the developer.

The open-source middleware ROS [16] is a software solution that eases the building process of robotic applications. It runs on top of Linux and has been gaining popularity in the robotics community over the last years. Different aspects of the ROS community are measured and reported yearly[6]. Among these, the total ROS packages downloaded are individually taken for each year and shown in Figure 1.1. Besides, not only the research community has shown its interest but also the industry. A consortium integrated by worldwide companies from multiple sectors, such as automotive or aerospace, has been growing over the years to extend the advanced capabilities of ROS software to manufacturing. Currently, there is much effort focused on a new version of the middleware as ROS2 since the first version does not satisfy real-time requirements. Therefore, ROS2 is based on Data Distribution Service (DDS), which is a standard protocol used in industry that meets real-time constraints due to its various transport configurations (e.g., deadlines and fault-tolerance). The decrease of ROS package downloads in 2021 may be due to more developers are slowly migrating to ROS2. As ROS became the mainstream option for roboticists and there are already several works on integrating FPGAs with it, more details are given in Section 2.8.1. It not only answers what are the trends in robotics, from a software perspective but extends it, exploring the method proposed in the state-of-the-art to enhance ROS with FPGAs.

2.8.1 The Robot Operating System Enhanced with Field Programmable Gate Arrays

As discussed previously, ROS became the most popular middleware as it provides many open source packages supporting all kinds of robots, data processing, and planning algorithms. Some concepts are explained to understand better the contributions shown in this section, focusing on integrating FPGAs into ROS.

ROS defines *nodes* where computations are performed. They communicate among each other via *topics*, characterized by the *type of message* (as different data structures) they transport. Nodes can be *publishers* (produce and broadcast data) or *subscribers* (consume data to process), and a combination of both. ROS provides all the communication mechanisms and protocols for all nodes in a system to communicate. The flexibility of nodes and messages allows for the reusability of ROS components to deploy algorithms in a distributed software system quickly. They can be programmed in various languages, for which client libraries exist.

The traditional communication scheme in ROS is shown in Figure 2.11. Every time a new node registers in the system, it does it with the *master node* via XML-RPC[7]. During the registration's handshake, meta-data information to send/receive data over topics to/from other topics is also shared. When a node wants to send or receive information to/from another node, a direct connection is established but over TCPROS[8], which is a transport layer for ROS messages and services. It uses standard TCP/IP sockets for transporting message data.

Message types are defined by an Interface Definition Languages (IDL). Therefore, specific *code generators* for each programming language can take advantage of this to generate language-specific bindings for each message type. Figure 2.12 shows an example of ROS message and depicts the complexity of the data structure that can be achieved as multi-level nesting

[6] `http://wiki.ros.org/Metrics`
[7] Remote Procedure Protocol (RPC) which uses XML to encode its calls and HTTP as a transport mechanism
[8] `http://wiki.ros.org/ROS/TCPROS`

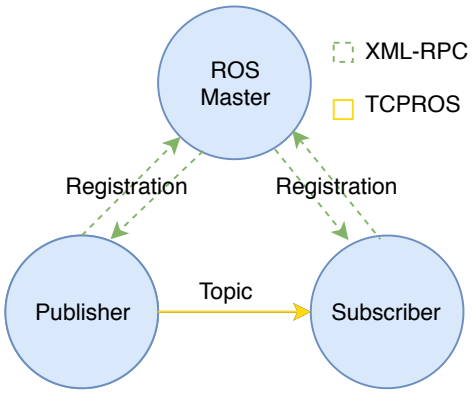


Figure 2.11: Basic ROS architecture

(message within a message) is possible. More details about the complexity of messages and how that is addressed in terms of hardware components is detailed in Chapter 5.

The industry has been paying attention to it with the arrival of ROS2, improving the quality of ROS1 by relying on industry standards, thus enabling more commercial use cases [67]. Each field will impose specific requirements for the robotic systems, such as performance, energy consumption, or real-time guarantees.

There have been different approaches to combine FPGAs with ROS. They can be grouped into three categories. The first one consists of focusing on a specific application. The second tries to generalize the concepts by proposing several frameworks, tools, and methodologies. The last one is based on the OS with support for reconfigurable systems, and ROS is integrated into it.

Application Specific

Some related works mainly focus on specific applications, accelerating some parts of a ROS-based software implementation. ROS is classically designed to run on a CPU which can also be combined with GPUs by enabling efficient management of data flow and shared memory. Lately, several works proposed to combine FPGAs with it [68, 69]. They mainly rely on Xilinx's

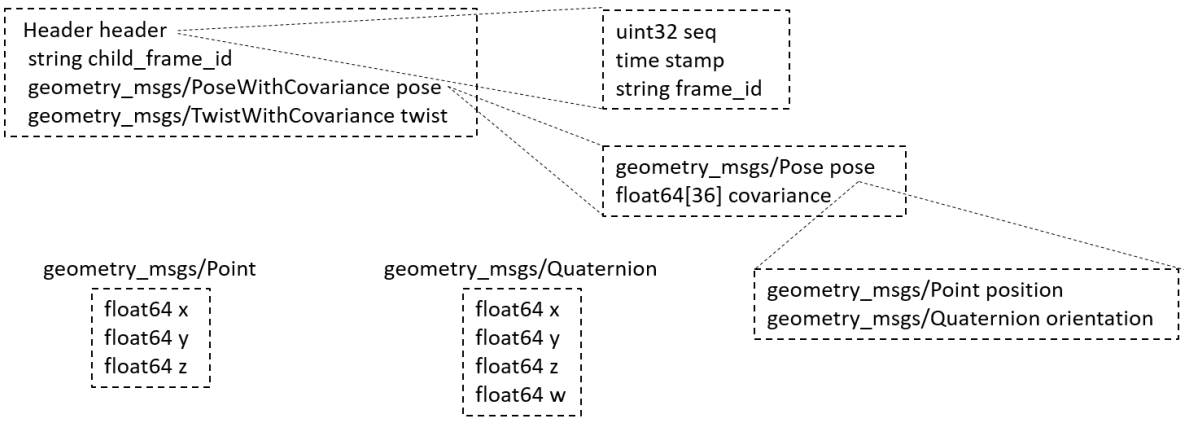


Figure 2.12: Complexity of ROS messages

SoCs FPGAs, which are capable of running Linux on their PS. Therefore, these designs cannot be realized in the absence of a PS capable of running an operating system as Linux. Queralta et al. [70] proposed a low cost 3D Lidar-based design. They rely on low-cost sensors to obtain 3D point clouds for localization and mapping algorithms. All the processing is implemented in VHDL. However, the communication with ROS, running on a PC, is done via the UART interface. Besides, they focus on a specialized design.

Frameworks, tools and methodologies

Some more generic approaches have been proposed, aiming to provide solutions for any ROS-based application.

Aldegheri et al. [71] presented a framework for the design and simulation of embedded video applications that integrate OpenVX standard with ROS. It does not target a single application but the specific field of image processing. It combines OpenVX, CUDA/OpenCL, and OpenMP to increase the embedded applications' parallelism and portability. Even though its portability, it is restricted to software implementations as it relies on a CPU (to run Linux) and the ROS API library to communicate with external systems.

Yamashina et al. [72] focus on *component-oriented* developments as a well-known method for reduction of costs in the development of software for robotics [73]. Hence, they propose FPGA-components as ROS-compliant ones, setting the following requirements: (A) the functionality of the ROS-compliant FPGA-component is equivalent to one implemented in software, and (B) the message type and data format used as the input and output of the ROS-compliant FPGA-component is equivalent to software implementations. This means that each ROS message type and data format used in ROS-compliant FPGA-component must be the same as the software ROS component. The authors rely on Xillinux[9], taking advantage of Xilinx's FPGAs-based SoCs (that include ARM processors). It allows effortless communication between PS and PL. The system is based on a file descriptor on the software side and a corresponding FIFO on the hardware side. Therefore, ROS is executed on the PS (running Linux), and the interaction between the hardware in the PL and ROS is straightforward via the file descriptor. The authors recognized the large amount of time and high development costs of these ROS-compliant FPGA-components and hardware IPs (HDLs-based) in general. Hence, they propose an automated design tool to improve the productivity of ROS-compliant FPGAs components [74, 75]. It is a design support tool that converts any targeted circuit (user logic in HDLs) into a component by giving a simple specification definition for data transfer [75]. The tool generates the interface circuit (hardware as HDLs files to interface accelerators) and software interface (C++ files). An input configuration file sets parameters such as bit width of communication channels, data ports for user logic, or transfer rates (input and output). Then, they are used to generate the corresponding interface via the artifacts mentioned previously. Like this, the integration of user logic is wrapped to easily integrate it into ROS via Xillinux. However, there is still the need to have Linux running on the PS to support ROS. Regarding the accelerators, the approach is shown with HLS techniques, but most probably, it would support HDLs-based accelerators with some changes to the automation tool. As far as the HDLs, they are the interfaces of the FIFO from Xillinux and a state machine to control the HLS accelerator.

[9] `http://xillybus.com/xillinux`

Ohkawa et al. [76] discuss a methodology to take advantage of these last two previous works [74, 75]. First, the Hardware/Software partitioning is done, like any HW/SW Co-Design. However, in this case, the partitioning is at the ROS level. This means that ROS nodes are split into multiple ones, all connected via topics. Like that, it is simple enough to identify which node will become a hardware accelerator to later on obtain the ROS-compliant FPGA-component.

Every new ROS node registers with the *master node* via XML-RPC[10]. During the registration's handshake, meta-data information is shared to send/receive data over topics to/from other topics. When a node wants to send or receive information to/from another node, a direct connection is established but over TCPROS[11], which is a transport layer for ROS messages and services. It uses standard TCP/IP sockets for transporting message data. Ohkawa et al. [78, 77] acknowledged the significant communication latency with the ROS-compliant approach mentioned previously, which also follows this connection scheme. They evaluated the latency introduced in the process of sending and receiving data as standard ROS publishers and subscribers running on an ARM core and from there to the ROS-compliant components. They concluded that for that approach to be functional in applications such as image processing, the way data is shared with the accelerators must have low latency. Therefore, they proposed to have an entire hardware implementation of the ROS Publisher/Subscriber communication scheme (for the TCPROS part, to exchange data directly between nodes) without the use of a SoC. The registration of nodes remains in software, which is done only once at the beginning. To achieve this, a TCP/IP stack needs to be available on the PL-side. The efficient hardware implementation SiTCP [79] was chosen. However, its drawback is that it provides the possibility to establish one connection at a time. Therefore, only the data transmission part is implemented into hardware, leaving the registration part on software. Consequently, this approach still relies on the PS, and it requires a technique to exchange data with the PL.

ROS was the leading middleware considered so far, mainly since it became the mainstream option for roboticists as shown previously in Figure 1.1. Other approaches have also been proposed targeting ROS but using other frameworks and tools as the center of their research, focusing on the issue of integrating hardware accelerators into ROS. Leal et al. [80] provide a tool that relies on the open-source PYNQ project [12] from Xilinx, which is also Linux-based. They automatize the generation of drivers to exchange data between PS and PL (only the data to be processed, not the entire ROS message). Therefore, it also generates a new ROS message type with only the payload of the ROS message that is transmitted to the hardware accelerator. Like this, it eases the integration of hardware accelerators but increases the complexity on the software side, as a bridge or interface to other ROS messages present in the system would still be needed.

Eisoldt et al. [81] focus on the integration of accelerators from the algorithmic point of view. Similar to previous references, there is a dependency on an embedded processor compatible with Linux. ROS runs on the PS, and the accelerated algorithmic calculations are on the PL. The authors take advantage of the shared memory between PS and PL and map the registers of the processing blocks to the node's virtual memory. This is done for data as well as controlling the start and stop of the processing blocks. So, they heavily rely on the memory management capabilities of the OS. However, for applications that require a large amount of data, only algorithmic parameters are mapped to memory, and data is streamed

[10] Remote Procedure Protocol (RPC) which uses XML to encode its calls and HTTP as a transport mechanism
[11] `http://wiki.ros.org/ROS/TCPROS`
[12] `http://www.pynq.io/`

over dedicated memory ports. There are references to specific HLS-related registers (e.g., AP_DONE, AP_CTRL), but no reference of *HDLs* accelerators is mentioned.

Ohkawa et al. [82] builds upon [77] to propose an HLS design flow for ROS protocol and communication circuit for FPGAs. It takes a definition of a ROS message, ROS related information (e.g., name of node, topic) and an application written in C++ for HLS to autogenerate an IP core. However, it is based on a hardwired TCP/IP stack that only allows one publisher per FPGAs. Their work can be considered a more general approach as it takes ROS definitions, despite being limited to HLS implementations.

In this dissertation, a generic architecture is proposed [21], to have a full hardware implementation without the need for any CPUs to close the gaps of these previous works. It is generic so that it can incorporate hardware accelerators designed in HDLs or HLS. Moreover, it leaves the possibility to replace the communication block if a different device is used or communications are handled by a CPUs if available or desired. In this case, FreeRTOS[13] was used to handle the communication between hardware accelerators (hardware nodes) and external ROS nodes, with an external Serial Peripheral Interface (SPI) device providing the TCP/IP stack. Similarly to [83, 84] (relying on the open-source lightweight IP (lwIP[14]) as the TCP/IP stack), an APIs provided functions to register/deregister hardware nodes as well as exchanging information between publishers and subscribers. However, there is no limitation concerning the size of data to be transferred in this dissertation (c.f., Chapter 3) [21] as in [83, 84], which was 512*KB* of data every 100*ms*. In case ROS is not the chosen middleware, and a different one is needed, the modular architecture allows replacing the specific-related middleware IP block thanks to the "plug&play" design. Lastly, it opens the possibility for robotic applications to use Dynamic Partial Reconfiguration (DPR), which can use the same hardware resources to implement different steps of a time-multiplexed algorithm. Consequently, the flexibility and power efficiency would be enhanced as well [85]. However, it still has an open point to generate the interfaces based on message specifications. This is tackled in Chapter 5, focusing on the generation of ROS and ROS2 components to integrate hardware accelerators into an FPGA-SoC [9]. An MDE approach is followed, with an extensive data-type analysis of ROS messages to generate VHDL components to produce AXIS frames to match the data representation of said messages. These components can be used together with Direct Memory Access (DMA) to exchange data between CPUs and accelerators. Like this, integrating accelerators into an already existing ROS system is aided with the tool detailed in Chapter 5. Lastly, Chapter 6 presents the workflow that generates the entire software/hardware architecture that includes multiple accelerators [25].

2.8.2 Operating Systems Support for Reconfigurable Computing

Lienen et al. [86] highlight the lack of a consistent programming model for implementing software and hardware functions. They close that gap by integrating ROS to ReconOS [87], tailored for multithreaded programming of hardware and software threads for reconfigurable computers. Similarly to [81], they also relied on the Linux virtual address space and shared memory to exchange data between PS and PL. In the follow-up work [88], they extended the support to partially reconfigure pre-allocated slots for either software or hardware executions based on callbacks.

[13] `https://www.freertos.org/`
[14] `https://savannah.nongnu.org/projects/lwip/`

2.8.3 Roboticists Interests

This summary shows that there is clearly increasing interest in the research community to provide tools and methodologies to attract roboticists to adaptive computing systems, particularly FPGAs. Three main characteristics would be considered by them:

- Tools & Methodologies [74, 21, 81, 80]
- Acceleration of internal ROS communication (i.e., interaction between nodes) [78]
- Optimization of ROS computational graph [76, 68, 69, 70]

A pattern is starting to emerge concerning the clear division of expertise as these groups focus on a specific topic. However, considering robotic systems as holistic ones, they are all tightly coupled.

Table 2.4 summarizes the main characteristics to consider at the time of integrating FPGA-based robotic systems into ROS. Even though middlewares help with integration aspects, designers still need to have -at least- some understanding of the low-level details concerning accelerators. That is why MDE can serve as a bridge between them. By using a general approach, designers can create simpler models that abstract away low-level details, while still being able to interconnect them and add new characteristics in each iteration. In the MDE line of thought, it is better to have a general approach in order to have simpler models and add new different characteristics in each iteration.

2.9 Model-Driven Engineering

MDE is described as the technique for using a staged model transformation process in which models are transformed in iterations [28]. It raises the level of abstraction, potentially circumventing any incompatibilities by abstracting middleware-specific characteristics. Besides, it helps non-experts to focus only on their areas of expertise (e.g., HW/SW Co-design, algorithms, control).

MDE focuses on creating and exploring domain models, which are conceptual models of all topics related to a problem-specific domain [89]. Concretely, a *model* is an abstraction of a system that often represents a partial and simplified view of a system (or a specific aspect) [31]. These models are usually more understandable and usually Platform Independent Models (PIMs). They can also be Platform Specific Models (PSMs).

Table 2.4: Integration of FPGAs and ROS.

Characteristics	[68, 69]	[70]	[71]	[80]	[81]	[82]	[86]	This Dissertation
Generalized Approach	✗	✗	✗	✗	✓	✓	✓	✓
Vendor Independent	✗	✓	✗	✗	✗	✗	✗	✓
CPU Independent	✗	✗	✗	✗	✗	✓	✗	✓
Handle Multiple Hardware Accelerators	✓	✗	✓	✓	✓	✗	✓	✓
Supports Multiple Middlewares	✗	✗	✗	✗	✗	✗	✓	✓
Generates Complete Architecture	✗	✗	✓	✗	✓	✗	✓	✓

MDE changed the paradigm from code-to-model-based development [90]. Models can be combined with automatic code generation techniques. Each additional information added to the final model is used to generate the desired code artifacts (e.g., C/C++, TCL scripts, VHDL). This new model-based paradigm allows to describe the application independently from a software and hardware platform, thanks to the levels of abstraction due to adding different aspects iteratively to the models. Different models can represent system elements in different domains and be part of the system's functionality, structure, or behavior. Additionally, MDE speeds up the development process and the formalization of such abstractions enabling the use of automated tools to verify the consistency of the generated artifacts, improving the reliability.

From the software side, MDE makes programming easier because low-level details are hidden behind abstractions that are easier to manage. From the hardware side, there has to be a description of the system concerning the low-level details that are hidden from the software side. Due to this complexity, the support of different languages and tools must be considered when following an MDE approach. For this, it is necessary to build conceptual descriptions of the systems to capture all the crucial characteristics for representing a formal model through a concrete syntax [31].

The development of advanced systems is challenging as expertise from multiple domains needs to be integrated conceptually and technically. Particularly for robotics, the main focus of the research community for software development based on models has been automatic code generation. One must consider that robotic systems involve several fields, so specific knowledge is required to combine all its constituting parts. There is a significant challenge for design, development, and implementation. MDE provides an efficient and flexible approach for developing robotics applications that copes with this challenge. By raising the level of abstraction, *models* become easier to understand, and it also simplifies the validation of the system. Another benefit of MDE is that it increases the level of *automation*, helping the process of code generation to bridge the gap between modeling and implementation [91]. However, the support for variability with regards to the targeted platform (e.g., embedded computers, FPGAs, middlewares) needs to be accounted for, namely PSM. Flexible model transformation and code generation techniques need to interface generated and non-generated artifacts, and models need to be adaptable to new information required by developers to cover all potential incompatibilities that could arise.

Generally speaking, from a software perspective, the field of robotics has been of great interest over the last few decades. Nordmann et al. [91] showed in a survey focusing on *design-specific modeling languages for robotics* how MDE has grown in the field since the 2010s. Even though the focus was on DSL, the preliminary analysis of the 137 publications gives an idea of the most relevant topics in the field. Over 53% of the surveyed literature focuses on the aspects of *Architectures and Programming* (c.f., Figure 2.13). The rest is divided into specific features related to robotics, such as kinematics (the motion of bodies in robotic mechanisms without taking the forces/torques causing the motion into account), sensing, and estimation or motion planning. This is of particular interest to understand which are the main aspects to be explored in robotic systems when focusing on MDE techniques to obtain artifacts to realize such systems. Understanding the techniques used in software fosters synergy among software, hardware and roboticists. In more detail, the subdomains from *Architectures and Programming* are characterized as follows:

- *Control & Handling of Events*: Organization of data and control flow as well as handling

of reactive and temporal events.

- *Architectural Structures and Viewpoints*: Description of architectural structures and software designs in general.
- *Distribution of Components*: Distribution of the software across the hardware, communication of components, and how middleware can be used to deal with heterogeneous software.
- *Architectural Styles*: Descriptions and guidance for the high-level organization of software providing a specialization of element and relation types, together with a set of constraints on how they can be used.
- *Concurrency*: Decomposition of software into processes, tasks, and threads, dealing with related issues of efficiency, atomicity, synchronization, and scheduling.
- *Interaction and Presentation*: Structuring and organization of interactions with users as well as the presentation of information.
- *Error and Exception Handling and Fault Tolerance*: Prevention, toleration, and processing of errors as well as dealing with exceptional conditions.
- *Families of Programs and Frameworks*: Software product lines or frameworks encapsulating commonalities among elements and targeting re-use by designing customizable components that account for variability.
- *Security and Safety*: Prevention of unauthorized access to and manipulation of information and other resources. Limiting damage, the continuation of service, speed-up of repair, and how to fail and recover securely.
- *Design Patterns*: Typically employed at a lower abstraction level than architectural styles.

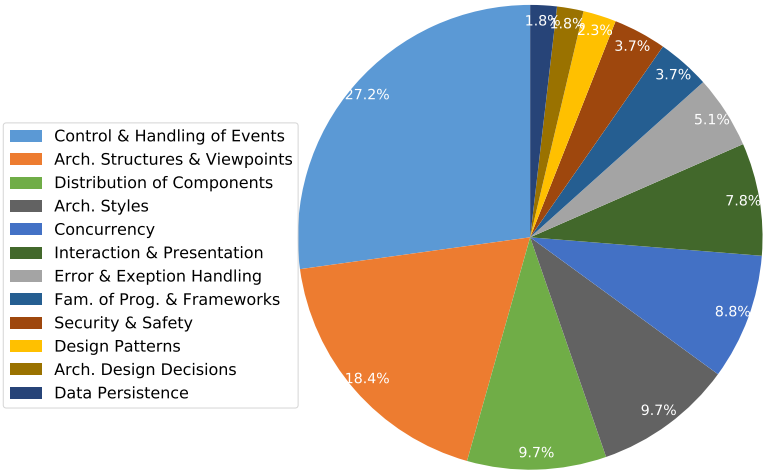


Figure 2.13: Aspects of architectures & programming of robotics in the state-of-the-art (based on [91])

- *Architecture Design Decisions*: Impact of quality attributes and the trade-offs among competing quality attributes that provide the basis for design decisions.
- *Data Persistence*: Handling of long-lived data.

The distribution of publications of each subdomain from *Architectures and Programming* is shown in Figure 2.13. It can be seen how the first 5 ones, namely *Control & Handling of Events*, *Architectural Structures and Viewpoints*, *Distribution of Components*, *Architectural Styles* and *Concurrency* cover over 74% of the topics. This helps to see their relevance of software in robotics. These domains overlap with hardware research, as shown in Section 2.7.

The following sections analyze how robotics benefited from embedded systems, namely GPUs and FPGAs, and how MDE improved the workflow development of these systems.

2.9.1 Control and Handling of Events

Willenberg et al. [92] proposed a framework to generate VHDL modules from behavioral models. They consist of hand-written kernels with aggregated buffered data-flow structures. The main goal is to obtain hybrid C++ and VHDL systems by bridging programming models and interfaces with activity diagrams, modeling data, and control flow. They opted to include synchronous FIFOs to avoid complex flow control in pipelined designs with generic control logic to protect it from illegal reads and writes.

Trabelsi et al. [93, 94] extended [95] by proposing a control design approach for FPGA-based reconfigurable systems. The approach is based on a semi-distributed control model. It splits different control concerns (monitoring, decision-making, and reconfiguration) between autonomous modular controllers. The splitting reduces the control design complexity and facilitates design verification, reuse, and scalability. According to the authors, the distribution of the control problem compared to a centralized one facilitates reuse as the latter is tightly dependent on the implemented design. Transitions in a centralized decision-making system depend on a global view of the system. Therefore, the whole decision model must be rewritten to include other reconfigurable regions. A new controller for each new region is only needed with their semi-distributed decision-making model. Their approach is based on Modeling and Analysis of Real-Time and Embedded Systems (MARTE), and due to following MDE techniques, it allows them to hide low-level technical details from designers and to automate code generation from high-level models.

Wiśniewski et al. [96] proposed a method for prototyping control systems with the possibility to include DPR. It is based on Unified Modeling Language (UML) state machine diagrams which are transformed into Finite State Machines (FSMs). There, each state can be either static (non-reconfigurable) or reconfigurable. A model-to-code process follows to generate VHDL files, which are used for synthesis, implementation, and the generation of bitstreams, including those for DPR.

Estivill-Castro et al. [97] recognized the difficulties in the semantics of UML that prevent FPGA implementations of complex real-time systems' models. This issue arises because UML concerns about *how to describe* a system rather than *building it*. Logic-labelled FSMs (LLFSMs) provide executable models with precise and defined semantics [98]. Hence, the authors proposed adapting LLFSMs with time-triggered semantics suitable for FPGAs. This approach enforces determinism for massively parallel, communicating LLFSMs. Models of

complex real-time behavior with perfect knowledge of timing requirements at design time can be implemented due to deterministic timing for each state. Furthermore, time-triggered deterministic behavior allows communication between multiple FSMs without race conditions or complex synchronization mechanisms.

Riché et al. [99] highlights that most algorithm experts do their design using floating-point without considering the limited resources available in FPGAs. For this, a fixed-point representation is always advised to deploy such algorithms in FPGAs. The authors presented an MDE-based tool as part of the LabVIEW NXG FPGA module to aid experts in obtaining fixed-point algorithms. The tool works on executable models built on the graphical dataflow model of computation "G". The novelty in their approach is that rather than analyzing the algorithm, data from a testbench is used to suggest fixed-point types within constraints provided by the user.

2.9.2 Architecture Structures and Viewpoints

Baklouti et al. [100] presented an MDE approach for Single Instruction Multiple Data (SIMD) SoC designs, based on UML and MARTE. The workflow consists of application programming, system modeling, deployment, and implementation generation. Like this, it is possible to generate a SIMD configuration at RTL from a high-level model. This also facilitates rapid prototyping and generation of different SoC for the exploration of configurations that best fit the requirements of the targeted applications. The different configurations can also be simulated as the artifacts, in this case, are VHDL files. The communication among all PEs is handled by a NoC in a 2D mesh topology.

Teodorov et al. [101] focused on the low-level flow of FPGA design. They proposed an MDE approach to model the physical synthesis process, focusing mainly on reconfigurable architectures. The authors argue that even though HLS has adopted the MDE methodology, physical synthesis is a complex resource allocation problem, which supposes a more complex model transformation. The core of their solution is to create a physical design automation Computed Aided Design (CAD) flow. This allows separating the application and design from the software tool for design and implementation. By allowing a precise specification of concepts and relations between them, MDE helps to reduce the complexity of developing and maintaining physical design tools. For this, they presented a generic metamodel[15] for describing hierarchical interconnected systems with arbitrary abstraction levels. As in similar publications, Design Space Exploration (DSE) is improved, enabling algorithm re-utilization.

Medeiros et al. [102] discuss a PIM to PSM converter. The modeling is based on MARTE, and the PSM are used to synthesize for FPGA implementations. MARTE is the chosen modeling language because it provides a clear distinction between hardware and software models compared to SysML. Moreover, real-time aspects can be modeled, which is important for embedded systems. In this case, DSE is also possible thanks to the proposed converter. It allows users to generate a set of embedded system configurations with particular needs for different hardware resources.

Leite et al. [103] aims to support the automatic generation of VHDL modules from a high-level specification of embedded systems. They propose a set of *mapping rules* to convert MARTE models into synthesizable VHDL description. The model-to-code transformation allows code

[15] A metamodel is a model of a model (i.e., a simplified model of an actual model of a circuit, system or software)

optimizations which result in an improvement of FPGA area consumption as well as system performance. Note that the proposed work supports synchronous and asynchronous method calls from sequence diagrams, which was a feature missing in the literature. In this case, the high-level model is converted into a PIM as a VHDL module. They extended the approach in [104], focusing on Aspect-Oriented Software Development. Here, a new set of mapping rules has been created for the model-level aspects to match their corresponding VHDL statements.

Zhang et al. [105] presented a toolkit to facilitate the design of complex asynchronous embedded systems with hardware and software components. The graph-based modeling approach allows validation through simulation and a more straightforward system construction. VHDL and C artifacts are generated for hardware and software respectively. The characteristics of the heterogeneous behavior of hardware and software are in a unified co-design model. The toolkit solves the challenge of extracting them, as the behavior of hardware modules for synchronous applications is usually controlled by a hardware clock, and the timing of a software module usually depends on the size and complexity of the code. Hence, a scheduling mechanism to keep the timing consistent and in sync among different hardware and software modules is also generated.

Streit et al. [106] proposed an automation flow to explore different hybrid hardware and software FPGA implementation from MATLAB/Simulink models. The novelty in the approach compared to the related work is the joint modeling of hardware and software *within the same* Simulink model for the generation of a holistic design. The difference is that an individual model for every component is needed in Simulink. That means two separate models for the hardware and one for the software. The approach would facilitate automatic DSE. They bridge Simulink with HLS and Vivado tools via CMake[16], in order to have PIMs. The code generated by Simulink has to be customized for hardware implementations due to its specific structure. Hence, code-based optimizations and the MDE-based model transformation are performed. The AXIS protocol achieves high-speed data streaming for inter-block communication.

Enrici et al. [107] proposed an approach to compiling system-level models into standard C code to optimize memory footprint. software implementation for a DSP platform as well as hardware accelerators are generated from this optimized C code. The base of this research focuses on the fact that multi-processor architectures raise the need to increase the level of abstraction of software paradigms. Besides, code generation from model-based specifications is considered to be more efficient than traditional paradigms where software is developed from code. In this case, the model is based on UML/SysML, and a model-to-code process is involved in obtaining an intermediate representation. The intermediate representation is used to optimize the system's memory footprint to produce the C code for the memory allocation and scheduling of signal-processing operations.

2.9.3 Combined Control and Handling of Events with Architecture Structures and Viewpoints

Vidal et al. [109, 108] proposed a design methodology to model DPR in UML. It targets multiprocessor systems. The first goal is to optimize the area through DPR. The second one is to increase the flexibility of the resulting system. Area optimization is achieved by

[16] `https://cmake.org`

reconfiguring co-processors connected to embedded ones. Flexibility is done by dynamically changing the behaviors of the co-processors at runtime. The proposed modeling approach, based on MDE techniques, intends to aid non-FPGA experts to include DPR in their designs. In this case, the artifacts are a set of bitstreams, including the one for the base design and the partial ones.

Ochoa et al. [110] proposes an approach based on MARTE, exploiting the capabilities of IP-XACT to model and automatically generate DPR SoC designs. IP-XACT is an XML format that defines and describes individual, reusable electronic circuits facilitating their use in creating integrated circuits. The aim is to obtain HDL code from high-level models of the system description. In this case, IP-XACT is used as an intermediate model to configure the accelerators for DPR and to automate the system integration. This work was extended in [111] to permit the verification of the platform description at different stages in the development process.

Trabelsi et al. [95] aims to increase the design productivity of FPGA-based reconfigurable systems. They do so by combining control distribution and high-level modeling to decrease the complexity and improve reutilization and scalability. Similarly to [93], control aspects such as monitoring, decision, and reconfiguration are distributed among individual controllers. However, a coordinator is used to keep the global system's constraints. The high-level modeling is based on MARTE. The proposed approach allows modeling adaptation aspects at different design levels (i.e., application, architecture, allocation, and deployment).

Corre et al. [112] focused on the lack of underlying platform architectures for FPGAs and proposed architecture models based on templates (source-code like sources that can be expanded according to the requirements). They then provide pre-parametrized designs according to the target domains (e.g., DSP, video). The approach improves reusability, code generation, and performance estimation. This last point is of particular interest to perform DSE. Hence, the designer can combine the parameters in the templates with the constraints for the expected design. Once a result satisfies the expected tradeoffs between cost and performance, the synthesized code for the hardware platform, the adapted software code of the application, and the project files corresponding to the FPGA backend tools are generated.

Ecker et al. [113] presents an automated process to generate hardware and software for embedded systems following OMG's Model-Driven Architecture (MDA)[17]. Their MDA adaptation consists of splitting the translation process into multiple layers. It starts with a formalized specification transformed into code. The code is then compiled (software) or synthesized (hardware) to finally be assembled into an embedded system design. The process is split into three layers. Model-of-Thing (MoT) represents the formalized specification. Model-of-Design (MoD) contains the implementation architecture (PIM). Model-of-View (MoV) are the PSM implementations. The process consists of translating MoT into MoD to translate it into MoV for code generation. The translation between models is done based on templates.

As mentioned previously in Section 2.8, different robotics middlewares have emerged over the years (e.g., YARP, OROCOS). This could become an issue due to incompatibilities of datatypes between different middlewares [114]. Many approaches also developed types-libraries with common comparable semantics but using different approaches for IDLs, serialization schemes, and APIs. On the one hand, Costa et al. [115] proposed the use of model-driven engineering concepts to *develop* specialized middlewares for particular application domains. The approach centers on building blocks, as meta-models, used to create models to specify

[17] `https://www.omg.org/mda/`

the configuration of the targeted middleware. The authors showed the feasibility of their approach with four domain-specific applications. On the other hand, since multiple robotics middlewares are available, Wienke et al. [114] proposed using model-based techniques for component reusability. They addressed data type compatibility in a structured way by developing a generic meta-model capable of representing data types from different middlewares and their relations. This is possible because the middlewares produce and consume serialized data to be streamed over a network. The meta-model describes data from various robotics middlewares in an abstract and unified way, but including the variables to be serialized. That model is used to generate serialization code to reuse the existing data-types of different middlewares. Moreover, they evaluated the features commonly found in different IDLs, which served as the base for the evaluation presented in Chapter 5 (cf. Table 5.1).

Table 2.5 summarizes these publications about MDE and FPGAs. Note that the two categories "Control & Handling of Events" and "Architecture Structures & Viewpoints" (c.f. Figure 2.13) cover 45% of the literature in the software-related state-of-the-art. It can be deduced from the literature shown above that these two categories are also the most relevant topics in adaptive computing systems, particularly FPGAs. It can be inferred that the primary purpose of these works, and in general for MDE, is to aid non-experts in a field (FPGAs in this case) to obtain the desired implementation. This is why either bitstreams or source code (VHDL or C) are generated from a high-level representation of the system, usually in UML (or derivatives such as MARTE). The main reason, according to the authors [94, 96, 108, 95, 110, 109] is to facilitate the workflow, which is usually cumbersome, especially for non-FPGA developers. Also, 25% of the publications analyzed in this work already support DPR, which is always a tedious process to do manually.

The main *takeaway* for this section is that MDE is a helpful modeling and design methodology to circumvent the arduous process of designing FPGA systems, whether they target organization of data and control flow as well as designs from an architectural viewpoint (system level). This has been proven to be successful, which is the main reason for this work, to now incorporate such techniques into robotic systems on FPGAs, considering the combined complexity of hardware design as well as the ones from the robotics field. Integrating FPGAs into ROS (Section 2.8.1 and Section 2.8.2) is an active research area as well as MDE techniques for code generation of FPGA-based systems Section 2.9. However, these two are mostly explored individually, opening possibilities for future research on how to combine their proven benefits to aid the workflow of FPGA-based robotic systems.

To conclude, Section 2.7 onwards introduced the state-of-the-art, in which the works presented showed different MDE techniques to facilitate the design process of FPGA-based systems. However, most of them focus on particular solutions or accelerators rather than on integration aspects. The techniques discussed previously are used to generate a system according to specified requirements. However, further specifications analysis is required to understand the system and determine its low-level details fully. These techniques are often only partially suitable for this analysis, and a technology that interprets the specified system is preferred. MARTE, the UML profile for modeling and analyzing real-time and embedded systems, is the most popular in the literature. MARTE supports modeling the timing behavior of FPGA-based systems (clock skew, jitted, and propagation delay). Furthermore, there is support for code generation and verification of FPGA-based systems. Therefore, it allows developers to model the behavior of hardware components and generate code for the hardware IPs. However, extra tools are needed before generating all the desired components to achieve the three objectives listed in Section 1.2. The ones proposed in this dissertation

Table 2.5: MDE approaches for FPGAs

Reference	Architectures and Programming		Models			Artifacts	DPR
	Control*	Architecture+	Levels	Language	Framework		
Trabelsi et al. (2014) [94]	✓	✗	PSM	UML, MARTE	GASPARD2	VHDL	✓
Riché et al. (2019) [99]	✓	✗	PSM	-	LabVIEW NXG FPGA Module	VHDL, Verilog	✗
Willenberg et al. (2010) [92]	✓	✗	PSM	UML	DMOSES/Eclipse	VHDL	✗
Wiśniewski et al. (2017) [96]	✓	✗	PSM	UML	-	Bitstreams	✓
Estivill-Castro et al. (2018) [97]	✓	✗	PIM	UML	-	VHDL	✗
Vidal et al. (2011) [108]	✓	✗	PSM	UML, MARTE	-	Bitstream	✓
Trabelsi et al. (2013) [93]	✓	✗	PSM	UML, MARTE	GASPARD2	VHDL	✗
Baklouti et al. (2011) [100]	✗	✓	PIM	UML, MARTE	SIMD Framework	VHDL	✗
Streit et al. (2018) [106]	✗	✓	-	Simulink	Matlab	Bitstream	✗
Medeiros et al. (2012) [102]	✗	✓	PIM, PSM	UML, MARTE	Papyrus	Bitstream	✗
Leite et al. (2014) [103]	✗	✓	PIM	UML, MARTE	AmoDE-RT	VHDL	✗
Zhang et al. (2016) [105]	✗	✓	PIM	SyncBlock	Tsmart-Edola	C, VHDL	✗
Enrici et al. (2018) [107]	✗	✓	PIM	UML	DIPLODOCUS	Bitstream	✗
Teodorov et al. (2011) [101]	-	✓	PIM, PSM	Generic metamodel	-	Bitstream	✗
Leite and Wehrmeister (2014) [104]	✓	✓	PIM	UML, MARTE	AmoDE-RT/GenERTiCA	VHDL	✗
Trabelsi et al. (2012) [95]	✓	✓	PSM	UML, MARTE	GASPARD2	C, VHDL	✓
Ecker et al. (2019) [113]	✓	✓	PIM, PSM	Custom metamodel	Infineon	C, VHDL	✗
Ochoa et al. (2011) [110]	✓	✓	PSM	UML, MARTE, IP-XACT	-	VHDL, C, SystemC	✓
Vidal et al. (2010) [109]	✓	✓	PIM	UML, MARTE	-	C, VHDL	✓
Ochoa-Ruiz et al. (2015) [111]	✓	✓	PIM	RecoMARTE	FAMOUS design	C, VHDL	✗
Corre et al. (2013) [112]	✓	✓	PIM	Kahn Process Network	Custom template based	C, Bitstream	✗

*Control & Handling of Events — +Architecture Structures & Viewpoints
Platform Independent Model (PIM) — Platform Specific Model (PSM) — Dynamic Partial Reconfiguration (DPR)

are described in Chapters 5 and 6. They are needed to perform data-type and data-flow analysis (described in Chapter 6), which is why *JastAdd* is more appropriate than frameworks like MARTE, as it is used for constructing compilers and tools alike.

Choosing one or the other does not mean they are not complementary. There might be cases where both frameworks can be used together. They are two different types of frameworks that serve different purposes. While MARTE is a UML profile for modeling and analyzing real-time and embedded systems, *JastAdd* is a metacompiler framework for implementing compilers and related tools, which is more fitted for the objectives of this dissertation. It allows to obtain a model of the system from a specification to be analyzed with MDE techniques and derive necessary information to further generate the components specified, validate them, and deploy the entire system automatically. All details to achieve these are explained in the following chapters.

3 Modular Hardware Architecture

Components within a heterogeneous distributed system, such as a robotic platform, usually exchange data. They can either generate or consume it; in this work, they are called *publishers* or *subscribers* respectively. As the goal is to combine FPGAs with robotic systems, hardware accelerators as publishers and subscribers exchange data with external software components in a distributed system, as shown in Figure 3.1. The main requirement is that external components do not make any distinctions with hardware accelerators, so they can be interchangeable if needed.

The aim here is to take advantage of the freedom and versatility that FPGAs provide for power-demanding applications to be part of any ROS (or any other middleware) architecture in an efficient way so that they can be seen from other ROS nodes as a regular one. This means that they behave like them in the way that they can send data as a publisher or receive data as a subscriber. This will allow deploying any given application, whether it is a heterogeneous system like a robot with multiple sensors and actuators or a distributed system such as a group of robots collaborating together, into *any ROS system*.

Most elements in the proposed hardware architecture are foreseen to be on the PL side but are not limited as some functionalities can reside on the PS side or external peripherals (e.g., communication from/to outside the FPGA [21]). The blocks shown in Figure 3.1 can be classified as:

1. `Accelerator-related components` comprising the base architecture needed to handle multiple hardware accelerators. Their design is only affected by the number of accelerators in the system.
2. `Message-dependent components` refer to those that follow a specification required for accelerators to be incorporated into the distributed systems.

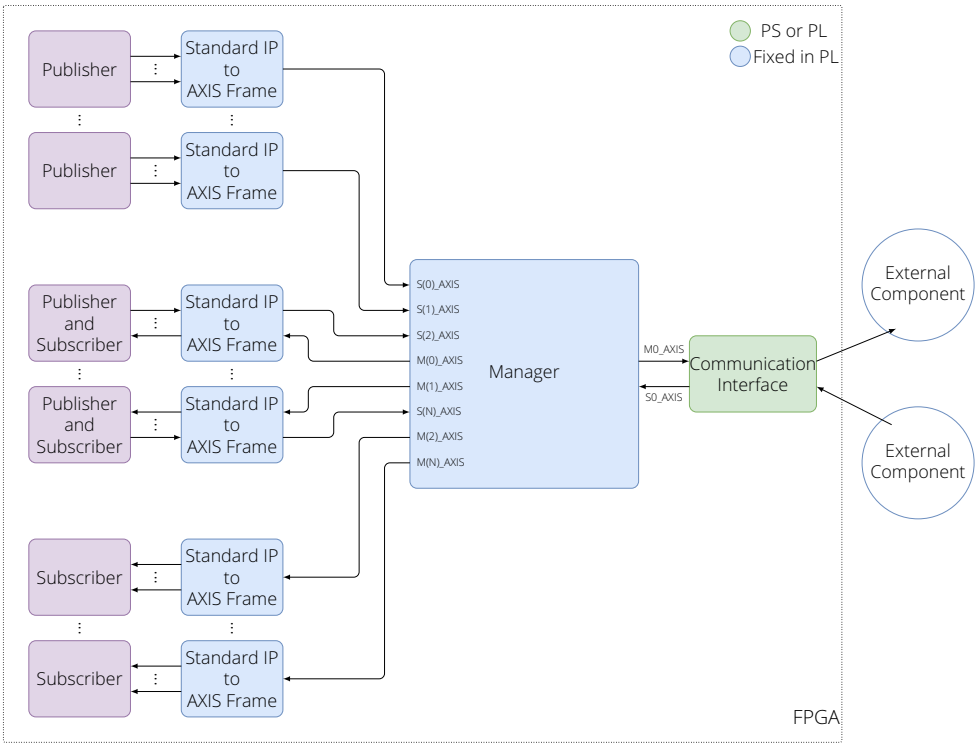


Figure 3.1: Generic base architecture[1].

3.1 Challenges and Goals

The main goal of the proposed modular architecture is to integrate hardware accelerators into robotics systems to enhance their efficiency. The first challenge is ensuring that the architecture components are extensible and reusable. The architecture must provide modularity to accommodate a wide range of hardware accelerators to achieve this. Furthermore, its components should be designed to allow for the easy integration of new accelerators without requiring significant changes to the overall architecture.

The second challenge is ensuring the proposed architecture is application-independent, allowing it to adapt effortlessly to new requirements. This requires the development of a standardized interface between the hardware accelerators and the software applications. This interface should be well-defined and independent of the specific application, enabling the system to support a wide range of applications without requiring significant modifications.

This chapter proposes a modular hardware architecture that leverages the principles of modularity, reusability, and standardization to address these challenges It is designed to provide a flexible and extensible architecture that can be easily customized to meet the specific requirements of different robotics applications. The following sections introduce the different components designed to achieve these goals.

[1]Accelerators can have intra-FPGA connections (omitted to have a simplified diagram).

3.2 Accelerator-Related Components

They are responsible for exchanging data between accelerators in FPGAs and external parts of the distributed system. Considering that most robotics systems follow a data flow graph-based-model design, the AXIS protocol is chosen as the standard interface among all blocks due to its simplicity and widespread usage. Additionally, if blocks or modules share the same interface, it can lead to different techniques such as DPR for efficient and low power designs [85]. The main component is the `Manager` block (Figure 3.4). The entity port (input and output signals) and the behavior of the `accelerator-related components` depend on the number of publishers and subscribers in the design, which is different for each use case. One could easily neglect one parameter by mistake as multiple modules are involved, raising the need to automatize their generation.

The `accelerator-related components` could have been simplified by relying on Xilinx's *AXI4-Stream Interconnect* IP core. However, this would imply becoming vendor-dependent, which reduces the possibility of porting to other vendors or platforms (e.g., Intel, Microsemi). Moreover, Xilinx's IP Core has a maximum of 16 interfaces per instance. Its resource utilization for 1 master and 4 slaves (16 bits for `TDATA`) is around 150 LUTs and 300 FFs for the simplest configuration. A similar solution following the approach presented in this work uses 364 LUTs and 67 FFs but four times the number of slaves, allowing for managing more accelerators with roughly a similar resource utilization[2].

The block handling the communication (`Communication Interface`) is inspired by the TCP/IP Five-Layer Network Model (Figure 3.2), where individual layers are adapted according to different needs. On the one hand, the Application (protocol), Transport (TCP or UDP), and Network layers (IP) will depend on each application. On the other hand, the Data Link (Ethernet or WiFi) and Physical (10 Base T or 802.11 standards) layers will depend on the device used for communication (e.g., onboard interface as Ethernet in case of some development boards, external devices such as the ESP32 providing Ethernet or the WIZ820io for WiFi, both with SPI interface).

In this proposed architecture, using the PS is not mandatory but optional. This feasibility is shown in [21] where ROS was not running under Linux. The `Communication Interface` was in the PL as a SPI-controlled device handling communications over Ethernet with external

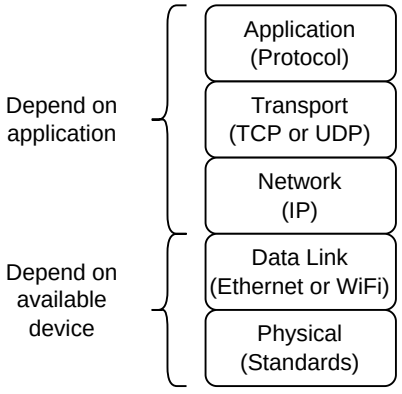


Figure 3.2: TCP/IP five-layer network model

[2]This numbers are for the Zynq-7000 SoC-FPGA (xc7z020clg400-1)

components. The versatility of the architecture is shown further in [9] by having native ROS running on the PS, acting as the link (on the software side) between external components and accelerators.

3.3 Messages-Dependent Components

They are the interfaces between hardware accelerators and external software components, which depend on message specifications (e.g., ROS `msg` format). The serial part of their entity is either AXIS master or slave, and the message specification determines the parallel part to interface. Each publisher or subscriber IP core needs its own block to convert its input and output ports (parallel) into an AXIS frame (serial).

A design simplification has been chosen, to use 8 bits for *data* and only use the minimum signals of the AXIS protocol (`TLAST` to denote the last byte in the transmission, `TVALID` and `TREADY` for handshaking). Reducing to byte widths allows to orient these blocks' design in a generic manner to ease the automatic code generation later on (Chapter 5). By doing so, each variable, regardless of its data type (e.g., int, float), is split into bytes to multiplex them individually. Variables (arrays or strings) and nested messages are transformed into an AXIS if their sizes are not fixed, relying on `TLAST` to denote their length. Figure 3.3 shows an example of an accelerator taking the role of a *publisher* with its interface component for the `sensor_msgs/Image` message specification (Listing 3.1). There it is possible to see that the fields *header*, *encoding* and *data* have been converted into AXIS while the rest have a bit-width according to their built-in data type. This message specification is used throughout the work as an example to highlight specific characteristics of the techniques shown, leading to an image processing use case.

Due to the design decision of using 8 bits for `TDATA` , a stream of bytes (AXIS frame) will be formed with the variables to interface. *startIndex* in Figure 3.3 depicts the position of the first byte of each variable in the resulting AXIS frame.

As it can be seen, message specifications can become complex data structures if they are formed by different data types, arrays with and without specified length, or even nested messages. Hence, manually writing their corresponding hardware components becomes a quite tedious and much more likely error-prone process. All their details, design decisions, and code generation is specified in Chapter 5.

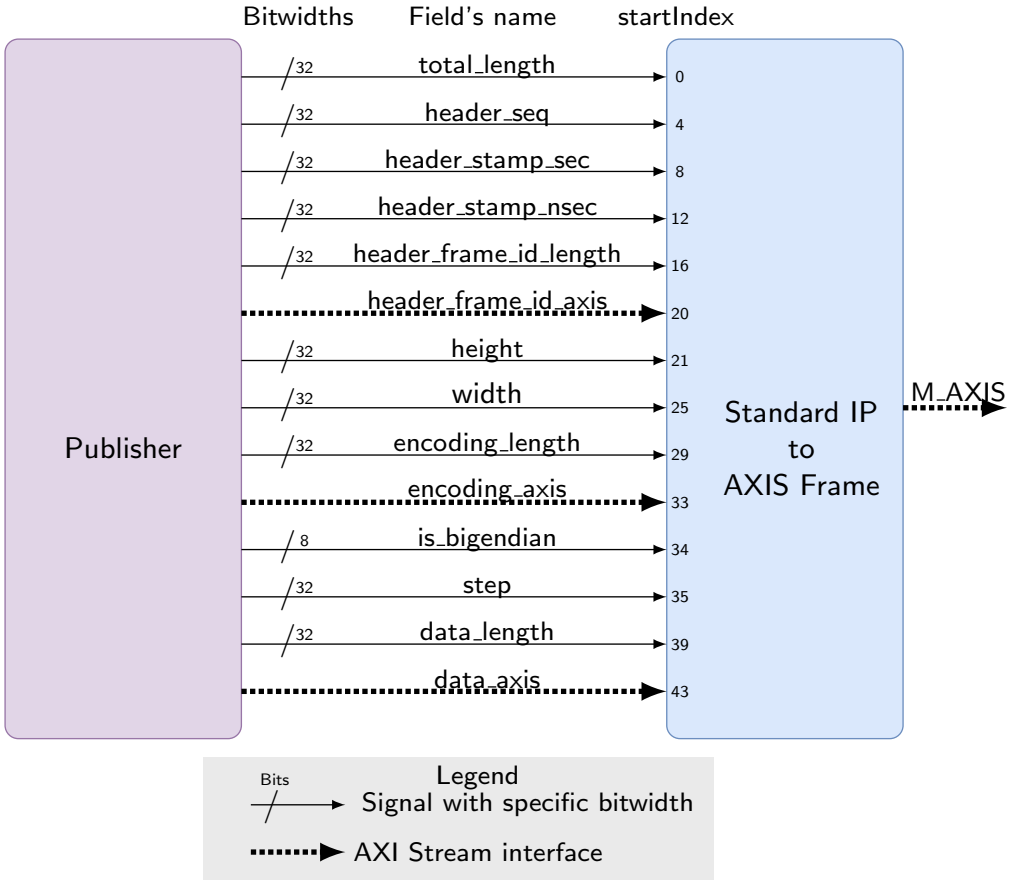


Figure 3.3: Hardware port for image msg

```
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```


Listing 3.1: ROS `sensor_msgs/Image` specification

3.4 Components of the Modular Architecture

There are four components in the proposed architecture, shown in Figure 3.1, and each of them are explained below.

3.4.1 Accelerators as Publishers and Subscribers

The accelerators within the architecture, those that perform the computation, can be of two types. Those that consume data to process it and those that produce data to be processed further. The former ones are defined as *Subscribers* and the latter ones *Publishers*. They can also be a combination of both, and their interfaces will be defined depending on their type.

3.4.2 Middleware-Based Hardware Interfaces

These are the components that convert from an AXIS frame to a message specification and vice-versa. Depending on the type of accelerator, they will be used by either subscribers or publishers. Subscribers receive data from DMA, so a conversion from AXIS to message is required. Publishers send data through the DMA, so they need to convert the message specification to an AXIS frame. These two types convert from parallel signals to a serial AXIS frame and vice-versa. All their details, design decisions, and code generation is specified in Chapter 5.

3.4.3 Manager

The `Manager` handles the proper communication between the two central parts of the system, namely the accelerators (as programmable logic) and the `Communication Interface` to external components. The `Manager` is composed of multiple blocks shown in Figure 3.4. Most of them are static, meaning they do not change their behavior. They have different purposes, which are explained below. Multiple options are proposed for the schedulers, which are explained in Chapter 4.

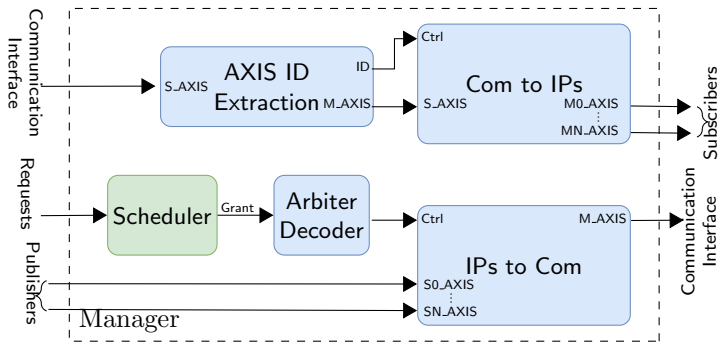


Figure 3.4: Manager

Communication

The `Com to IPs` and `IPs to Com` are the ones that route the AXIS frames from/to the `Communication Interface` to/from *Subscribers* and *Publishers*, respectively. As the AXIS uses one (8 bit) signal for data (`TDATA`) and three for handshaking (`TVALID` , `TREADY` and `TLAST`), they have to be multiplexed and demultiplexed because the communication is 1-to-N or N-to-1, depending on the direction.

On the one hand, the `Com to IPs` handles the communication from PS to PL. Hence, data for *all accelerators* comes from one DMA and needs to be *demultiplexed* to each accelerators. Hence, this module is composed by three demultiplexers for `TDATA` , `TVALID` and `TLAST` and one multiplexer for `TREADY` . On the other hand, the `IPs to Com` handles the communication from PL to PS. Therefore, data from each accelerator is *multiplexed* towards the DMA. So, this module is composed by three multiplexers for `TDATA` , `TVALID` and `TLAST` and one demultiplexer for `TREADY` .

The bit-width of the multiplexers and demultiplexers is determined by the number of accelerators, which will influence the bit-width of their control signal, computed with Equation (3.1).

AXIS ID Extraction

Each AXIS frame must include an ID to route it to/from the corresponding accelerator. The ID is inserted on the software side for the frames coming from the DMA (for subscriber IPs), and has to be extracted on the PL before it can be demultiplexed. Figure 3.5 shows an example of an AXIS frame streamed from the PS to PL that corresponds to a ROS message composed by an 8-bit integer array with two elements. The AXIS frame starts with the ID, "01" in this case. It follows the total number of bytes to be transmitted, six in this case. It ends with the array's length (two bytes) and the two corresponding integers. Details about the creation of AXIS frames from ROS message specifications are given in Chapter 5. The frames from the accelerators to the `Communication Interface` also include an ID, which is added by the converters, explained in detail also in Chapter 5.

Figure 3.5 shows a simplified version, omitting the signals for handshaking. Below the clock, the input, `TDATA` , shows the complete frame. The ID is extracted on the first clock cycle and latched to the output, which serves as the control signal for the following block (`Com to IPs`). The ID to be latched is converted from decimal to binary, and its bit-width is determined by Equation (3.1). Besides optimizing this block's resources, the bit-width needs to be adjusted to optimize the multiplexers and demultiplexers, where the input and output ports are also expanded or shrank accordingly to the total number of accelerators in the expected architecture. This is the main reason these components are called `accelerator-dependent` because their entities vary accordingly to the total number of accelerators involved in the architecture. The other output of the `AXIS ID Extraction` is the AXIS frame itself, which starts on the next clock cycle. As seen in Figure 3.5, this block does not introduce any delays.

$$ID_{bit-width}(N) = ceiling\left(\frac{log(N)}{log(2)}\right) \tag{3.1}$$

where N is the number of accelerators.

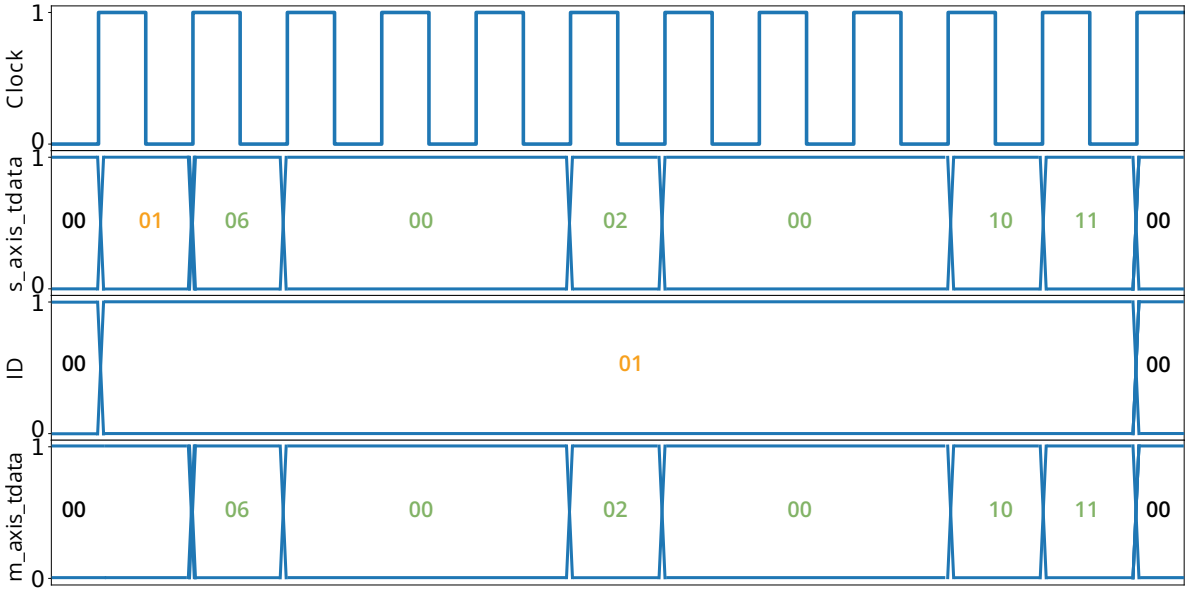


Figure 3.5: AXIS ID extraction signals

Arbiter Decoder

This block takes as input the grants from the scheduler. Therefore, its bit-width will be equal to the number of accelerators. Table 3.1 and Table 3.2 show two minimal examples of the decoders for inputs of two and four grants for accelerators. Because the control signal of the multiplexers or demultiplexers is in binary, this decoder is needed. Only one accelerator can receive the grant for a shared resource at any clock cycle, and the communication blocks have to select the corresponding signals accordingly to route the AXIS frame from/to the correct accelerator. As the multiplexers or demultiplexers in the `Communication to IPs` and `IPs to Communication` need as many bits as Equation (3.1) determines, this decoder is needed to convert from grants to control signals with the same number of bits.

3.4.4 Communication Interface

The synergy between software architectures and FPGAs is a challenging task. Usually, for robotic systems, the first one runs on an embedded computer, while the second one is a hardware platform on its own. Hence, one of the challenges is establishing communication

Table 3.1: Decoder with two input's truth table

Input Grant[1:0]	Output sel[0:0]
01	0
10	1

Table 3.2: Decoder with four input's truth table

Input Grant[3:0]	Output sel[1:0]
0001	00
0010	01
0100	10
1000	11

between both systems to integrate one another. This `Communication Interface` is then the one that establishes the connection between the components in the FPGA and external ones. The aim is to integrate accelerators as publishers or subscribers into existing robotic systems as standard components (i.e., ROS nodes), making no distinctions with external components. This means that they behave like them in the way that they can send data as publishers and receive data as subscribers. The use case in this work is based on the ROS communication protocols, but it can be generalized and adapted if needed.

The most basic architecture of a ROS system is composed of three *nodes*, which is where processes perform computations. *ROS Master* is aware of all existing nodes in the architecture and coordinates them accordingly. A *Publisher* can broadcast messages over a topic for other nodes to receive. A *Subscriber* can receive a message if a compatible topic is available. All the communication within ROS is done via TCP Sockets. They allow communication between applications, either on the local system or spread in a distributed TCP/IP-based network environment. There are two different actions involved in the communication between nodes. The first one is the registration or unregistration of a node. This only involves the ROS master and the node performing the action. The Remote Procedure Call (RPC) protocol XML-RPC, which uses XML to encode its calls and HTTP as a transport mechanism, is used. The second communication protocol is a transport layer based on TCP/IP sockets (TCPROS) to establish direct communication between nodes and transfer data. Initially, a subscriber requests from the master a given topic. If it exists, the master will inform the subscriber about the node publishing the requested topic and, subsequently, its IP address and port number. Afterward, the subscriber will contact the publisher directly to request the topic, and the connection between them will be established. Every time the publisher has new data to broadcast, it will send it directly over the earlier connection to the subscriber. At this point, the ROS master does not take any action as the connection is directly between publisher and subscriber.

As all the communication is based on TCP/IP, a publisher can have multiple subscribers. The publisher will communicate with the ROS master over one socket and listen to another one for any subscribers requesting a topic. When this happens, communication via a new socket is permanently established with each subscriber. Hence, a publisher will have $2 + n$ sockets, where n is the number of subscribers.

Hence, the `Communication Interface` has to be able to handle specific communication protocols (TCP/IP in this case) to integrate accelerators in an existing software solution. There are several ways to achieve this, considering the available resources on current FPGA boards and depending on the physical layer of the communication (e.g., Ethernet).

External solutions can be used if there is no available ethernet connected to the PL. As demonstrated in [21], an external SPI-Ethernet component was used. The design is based on the WIZ820io, a compact-sized module that includes a W5200 (MAC, Ethernet, and PHY layers plus a TCP/IP stack) and an RJ45 jack, controlled via SPI. The controller used is shown in Figure 3.6. The *AXIS Receiver* is the one that will receive the AXIS frame coming from the `Manager` containing data to be sent over TCP/IP and metadata needed to establish the TCP/IP communication (i. e., IP and port). Therefore, it will push the data into a FIFO and extract the metadata to send it directly to the *WIZ820io Interface* block. This one acts as an SPI master to send and receive proper commands to the W5200, which is the one managing the actual TCP/IP connection. The *AXIS Sender* block retrieves the incoming data from external components and sends it over AXIS to the corresponding IP. The FIFOs circumvent any transfer rate difference between AXIS (100*MHz*) and SPI Stream (50*MHz*).

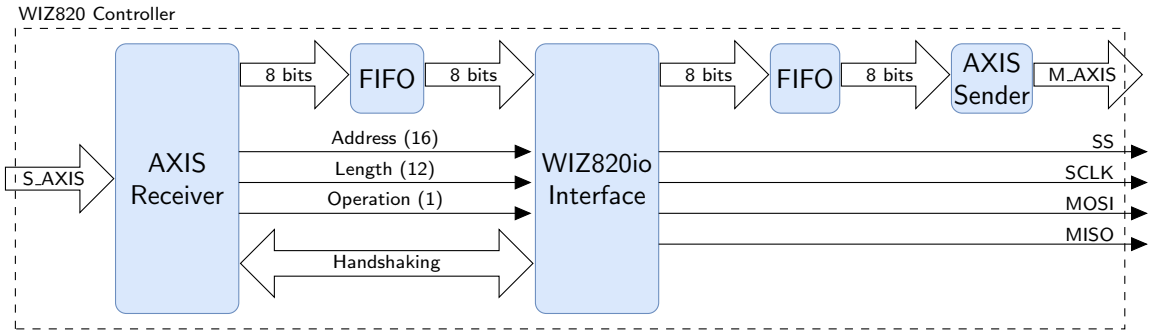


Figure 3.6: SPI master architecture with AXIS interfaces.

This is the first solution to show that the entire architecture depicted in Figure 3.1 can be entirely realized on hardware. However, there also exists the case that the ethernet is present, and in the case of most of Xilinx's development boards, it is routed directly to the PS. This opens up two possibilities. The first one consists of using a standalone application or relying on a real-time OS, for example, FreeRTOS, which is very well supported by Xilinx's tools. The second one, followed in [21, 9], consists of relying on the capabilities of the ARM processors to run an OS. This allows having Ubuntu and native ROS on top. However, there are some challenges to solve with this approach related to the exchange of data between PL and PS which is addressed by the hybrid hardware/software schedulers.

3.5 Evaluation

The evaluation of the modular architecture proposed in this chapter is split into two parts. The first one concerns all the blocks included in the `Manager`, which are the same for all designs as they only depend on the number of accelerators. The second one is for the proposed schedulers, which are, in fact, part of the `Manager`, but they not only depend on the total number of accelerators but the different algorithms. These have different consequences, and there are several metrics proposed to understand the behavior of each of them appropriately. All results shown here are after synthesis for the Xilinx's Zynq UltraScale+ xczu7ev-ffvc1156-2-e.

Table 3.3 shows the resource utilized by each of the components included in the *Manager* (Figure 3.4), which are common for all designs. Even though the `Manager` includes a scheduler, its different options are evaluated separately in Section 4.3. It can be seen that the `AXIS ID Extraction` is the only module that utilizes FFs. The reason is that it requires some registers to keep the ID of the frame for the entire time it is being streamed, so it is latched. The remaining modules do not require FFs as they are decoders, multiplexers, and demultiplexers which are purely combinational circuits.

Figure 3.7 shows the overall representation of how each of the components affects the resource utilization for the `Manager` as a whole. The `Arbiter Decoder` does not consume much LUT for a small number of accelerators, but the logic increases the more accelerators are in the system. Having more accelerators in the system means that the output port of this decoder and, therefore, its logic will increase. The total number of output signals is given by Equation (3.2), where the condition given by Equation (3.3) has to be met.

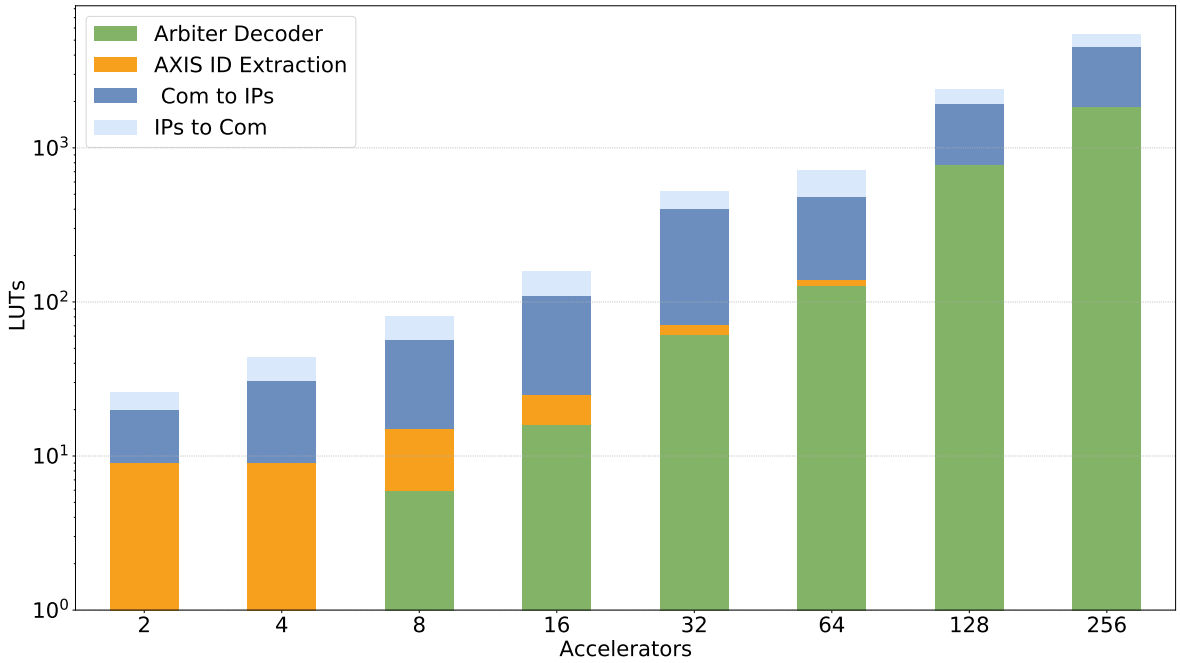


Figure 3.7: Resource utilization in common IPs inside the manager

$$Decoder\ Output = 2^N, \qquad N \in 1, 2, 3, 4, 5, 6, 7, 8 \tag{3.2}$$

$$Total\ Accelerators \leq 2^N \tag{3.3}$$

The LUTs for `AXIS ID extraction` are roughly the same for all number of accelerators. However, they influence the total resource consumption more when there are a few accelerators because the other components do not consume much.

Table 3.3 shows the LUTs and FFs utilized by the `AXIS ID Extraction` component. The LUTs do not increase when the accelerators do as the logic is the same. However, the FFs increase because the signals to latch increase when more accelerators are included in the design as

Table 3.3: Resource utilization in common IPs inside the manager

Accelerators	Arbiter Decoder		AXIS ID Extraction		Communication to IPs		IPs to Communication	
	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs
2	1	0	8	12	11	0	6	0
4	1	0	8	13	22	0	13	0
8	6	0	9	14	42	0	24	0
16	16	0	9	15	84	0	48	0
32	61	0	10	16	329	0	122	0
64	128	0	10	17	345	0	239	0
128	773	0	11	18	1138	0	478	0
256	1837	0	11	19	2644	0	946	0

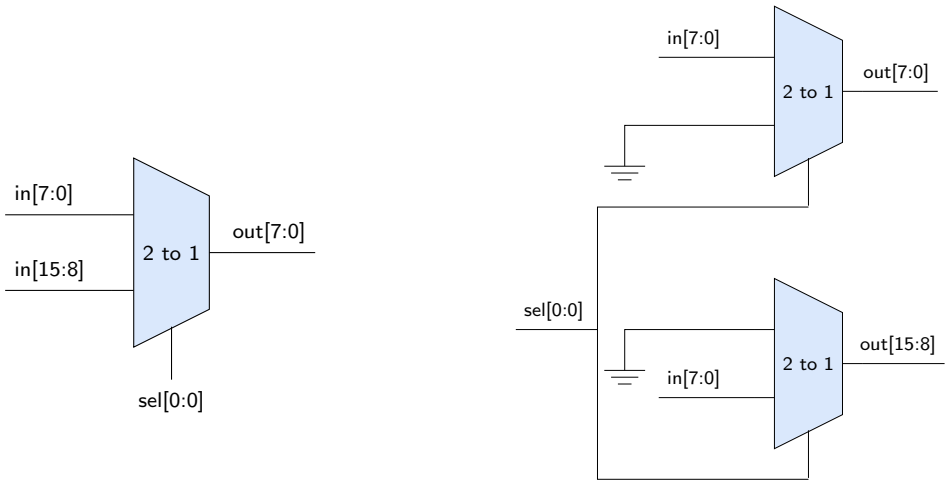
their binary representation requires more bits the larger the number is (c.f., Equation (3.1)). This is the only component that consumes FFs because it latches the ID, making this a sequential circuit compared to the rest which are only combinatorial circuits.

The two blocks concerning communication have a constant resource utilization relative to the number of accelerators in the design. However, the main reason for the doubling of LUT of the `Communication to IPs` block with respect to the `IPs to Communication` lies in the fact that the former one is formed by three demultiplexers and one multiplexer and the latter one of three multiplexers and one demultiplexer. The LUT will double because, based on Xilinx's CLBs which only contains multiplexers, to build a demultiplexer, two multiplexers are needed, as shown in Figure 3.8.

3.6 Summary

This chapter introduced the base architecture that serves as the playground for this entire work. It is a modular architecture to be used as the foundation for integrating FPGAs into existing robotics solutions. An explanation and reason for each component included in the architecture are detailed, as well as their need and how they interact with each other. The design of each component is generalizable, easing their code generation and automatic deployment of the entire system. The evaluation focuses on the resource utilization and scalability of the `accelerator-related` components, which are the ones that are affected by the total number of accelerators in each design where the modular architecture is used. It is worth mentioning that all results are shown for Xilinx's Zynq UltraScale+. However, the implementations described in this chapter (and this dissertation) can be ported to any other target device available from Vivado and other vendors as all components are vendor-independent and based on the VHDL-93 standard.

Details of the modelling, code generation and automatic deployment of the components described in this chapter are presented in Chapters 5 and 6.



(a) 2 to 1 multiplexer (b) 1 to 2 demultiplexer

Figure 3.8: Multiplexer and demultiplexer

4 Hybrid Hardware/Software Schedulers

Different scheduling algorithms are proposed to have different options to adapt to the needs of each application. A software implementation is needed to schedule the transactions from PS to PL, and hardware counterpart for PL to PS ones. The hardware implementation details are described below, as one needs to consider the low-level signals that are not needed in software. Particularly for this work, tasks represent the time each component can stream its data. In general, a scheduler has multiple inputs for the *requests* from the accelerators, meaning that they have data available to broadcast and they are ready to be scheduled. The output will be the *grant*, allowing only one accelerator to perform the transmission at any given time. An example is shown in Figure 4.8, and all the details are explained below.

4.1 Challenges and Goals

The schedulers play a critical role in the modular architecture described in 3, and are part of the `Manager`. As a result, they need to meet specific requirements regarding extensibility and adaptability to new applications. It is essential to achieve this to have a diverse range of schedulers available, which can be selected based on the specific needs of each application, including their extensibility (i.e., the number of hardware accelerators required).

The schedulers must be designed to be independent of both the hardware accelerators and the applications they are scheduling to ensure maximum flexibility. This means schedulers should be able to accommodate varying numbers of accelerators without requiring significant modifications. Ultimately, the goal is to provide a wide range of schedulers that can adapt to the specific needs of each application while remaining extensible and independent of the underlying hardware architecture.

As the AXIS is the chosen communication protocol, `TVALID` is used as *requests* and `TREADY` is used as *grants*. Each of the implemented schedulers shown below works as follows. Each accelerator that sets `TVALID` to one will get a *grant* as long as it is the only one that set a *request*. Only one accelerator can get the *grant* on each clock cycle. Therefore, it will be computed accordingly to each algorithm when multiple accelerators have data to stream (`TVALID` set to one) at the same time. The end of each task is denoted with `TLAST` , as dictated by the AXIS protocol.

There are four characteristics considered for the schedulers:

- ***Preemptive:*** a running task is paused when a higher priority task arrives and gets the grant. The first one resumes after the latter one completes.
- ***Non-Preemptive:*** this algorithm will not interrupt the currently executing task until the execution is terminated.
- ***Fixed Priorities:*** priorities are set at the start of the application and kept fixed for the entire runtime of the process.
- ***Dynamic Priorities:*** priorities are updated dynamically during runtime according to the scheduling algorithm.

Considering that the end goal is to generate all these components from an abstract description of the system, the core of the implementation of the different schedulers has to be generalizable. Therefore, the adaptable statechart shown in Figure 4.1 is used as the base for all proposed algorithms. It is composed of two types of states and transitions. Ones are static, common for all schedulers, and others are adapted for the needs of each algorithm, whether it is a specific computation (e.g., deadline, slack) or conditions for the transitions. Therefore, only certain parts of the statechart differ from one scheduler to the other. A *priority table* is initialized at the beginning. The algorithm-dependent conditions are computed to use them for updating the *priority table* accordingly, depending on the algorithm. The updated priorities are used in the *Set Grant* superstate to find the maximum value (highest priority) to asses which accelerator will get the grant. The transitions within this superstate also depend on the algorithm, as each of them dictates how to react to new requests or internal conditions.

There is no relation between multiple accelerators and the schedulers, so there is data independency for all tasks. As far as deadlines, multiple definitions exist:

- ***Implicit deadline***: when the relative deadline D_i is equal to the period T_i , i.e. $D_i = T_i$, for every task τ_i.
- ***Constrained deadline***: when the relative deadline D_i is not larger than the period T_i , i.e., $D_i \leq T_i$, for every task τ_i.
- ***Arbitrary deadline***: when the relative deadline D_i could be larger than the period T_i for some task τ_i.

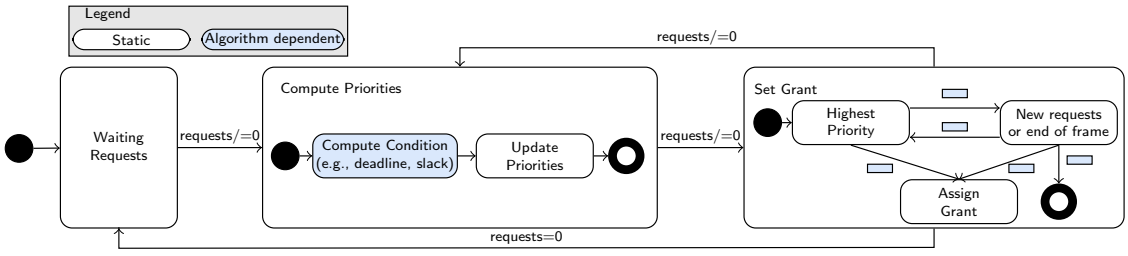


Figure 4.1: Adaptable statechart, generic for all scheduling algorithms

4.2 Scheduling Algorithms

Four different algorithms are proposed, two of them with two variations, making six schedulers in total. They are based on traditional software solutions with the corresponding adaptations to hardware implementations and the chosen streaming interface. *Implicit deadlines* are chosen for the Earliest Deadline Frist (EDF) and Least Slack Time (LST) algorithms presented below. Moreover, soft real-time constraints are assumed for the system, meaning that missed deadlines will not have catastrophic consequences.

4.2.1 Least Recently Utilized (LRU)

The Least Recently Used (LRU) algorithm is mainly used to manage buffer memories and caches. It *dynamically* changes the priorities based on the accelerator that got the grant *the latest*. This one will be moved to the bottom of the priority list, allowing *all* accelerators to get the grant. On the positive side, this guarantees that there will be no resource starvation. On the negative side, some accelerators will likely miss their deadlines. This is more evident the more accelerators are included because it takes *N* rounds (in the worst case) for an accelerator to be on top of the priority list. In order to mitigate this, only the accelerators which set their *requests* are considered each time the priorities are evaluated.

4.2.2 Fixed Priority (FP)

The Fixed Priority (FP) algorithm is a *static* priority one, meaning that priorities will remain unchanged during the entire execution time. Figure 4.2 shows a minimal example for the preemptive and non-preemptive versions. For the non-preemptive, each accelerator will complete its task before another receives the grant. For the preemptive, the accelerator with the lowest priority (*Acc1*) is preempted as soon as *Acc2* (with the highest priority) sets its request. Priorities are set based on the port defined in the entity of the module. As the goal is to have a *generalizable* design to ease the automatic code generation, all input ports (AXIS-related signals) are defined as variable-sized vectors. Hence, the position of each signal (acting as *request*) in the `TVALID [N]` array will dictate the priority of each accelerator. The lower the N, the higher the priority.

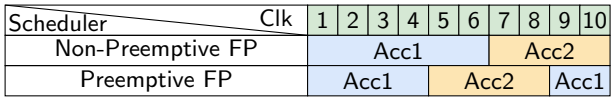


Figure 4.2: Fixed priority scheduling with and without preemption

This algorithm has two variants, namely, Preemptive Fixed Priority (PFP) and Non-Preemptive Fixed Priority (NPFP). The difference lies in whether the *grant* changes in every clock cycle. The computation's logic is only generated if needed to improve resource utilization. As the algorithm's name suggests, the *priority table* is never updated. It is initialized once, based on the width of the *requests* port depending on the number of accelerators (N), from 0 to *N* – 1. Accelerators are assigned indexes, and the one with the lowest index, which sets its *request*, is assigned the *grant*.

4.2.3 Earliest Deadline First (EDF)

EDF is a *dynamic* priority schedule. This means that priorities will change on each clock cycle for the entire running time, depending on the state of the *requests* and how close each deadline is with respect to the current time. Deadlines are decremented for each accelerator, with the request set to one on every clock cycle. Newly arrived requests are assigned for priority based on their implicit deadline. Figure 4.3 shows the behavior of the scheduler with two tasks and how *Acc1* is preempted when *Acc2*'s deadline is closer, so according to the algorithm *it must* run first in order to avoid missing its deadline. Once this one finishes, *Acc1* can resume.

Scheduler \ Clk	1	2	3	4	5	6	7	8	9	10
EDF	Acc1			Acc2				Acc1		

Figure 4.3: Earliest deadline first scheduling

Theoretically, EDF can achieve a 100% utilization (U) according to Equation (4.1).

$$U_{EDF} = \sum_{i=0}^{n} \frac{C_i}{T_i} \tag{4.1}$$

where C_i is the Worst Case Execution Time (WCET) of a task with period T_i, and n represents the total number of tasks in the system.

Particularly for EDF, two main computations are required. The first one is to decrement the deadlines of every accelerator to increase their priorities. As this is a dynamic priority scheduler, priorities are updated every time a new *request* arrives to update the *priority table*.

There are two versions proposed. On the one hand, the resources-optimized one (Resource-Optimized Earliest Deadline First (ROEDF)) follows the statechart shown in Figure 4.1. There will always be a two-clock difference when grants are assigned. On the other hand, the latency-optimized version (Latency-Optimized Earliest Deadline First (LOEDF)) is a slight modification of the generic statechart as some computations are merged into the same state. This will reduce the state transitions (leading to fewer clock cycles) but increases resource consumption. Therefore, a tradeoff between resource utilization and latency is detailed in Section 4.3.

4.2.4 Least Slack Time (LST)

LST is also a *dynamic* priority schedule. Contrary to EDF, this algorithm evaluates the slack time of the accelerator requesting the grant on every clock cycle following Equation (4.2)

$$s_i = d_i - a_i - c_i \tag{4.2}$$

where s_i is the slack time (priority) of the accelerator I with a deadline of d_i. The time acc_i sets its *request* is represented by a_i, and c_i is the remaining execution time for the task. The accelerator that holds the *grant* will get a_i incremented and c_i decremented by one on each

clock cycle to *keep its slack time* until it finishes its transmission or an accelerator with a higher priority preempts it. For all other accelerators with *request* set to one, their arrival time will be incremented by one, resulting in a lower slack time on each clock cycle.

Figure 4.4 shows how LST looks like with two accelerators, considering the changes in their *slack time* causing them to preempt each other.

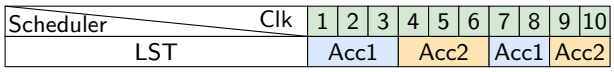


Figure 4.4: Least slack time scheduling

This algorithm is the most complex one as it needs to keep track of the remaining processing time, deadlines, and arrival time of new requests. All these have their own functions, which are translated into a higher resource consumption compared to the previous ones shown before.

4.3 Evaluation

The proposed schedulers are, in fact, part of the `Manager`, but they not only depend on the total number of accelerators but the different algorithms. These have different consequences, and there are several metrics proposed to understand the behavior of each of them appropriately. All results shown here are after synthesis for the Xilinx's Zynq UltraScale+ xczu7ev-ffvc1156-2-e.

The accelerators competing to get a hold of the DMA have two parameters. One is the *transfer time* (T), in this case, representing the length of its payload to be streamed in bytes (one byte per clock cycle is transmitted). The other is the *frequency* (F), the number of clock cycles after its last transmission, and the availability of new data to be streamed. These two definitions are adapted from software to hardware implementation with a streaming interface between accelerators and the schedulers. Each pair is called a set $S_i = \{T, F\}$, and the evaluation methodology followed for the schedulers consisted of a normal distribution for the generation of N sets for $M = \{2, 4, 8, 16, 128, 256\}$ accelerators. The evaluation was done until 256, but it is not limited as larger values can be used. The N-sets constitute a dataset $D_M = \{(S_1, \sigma_1), \ldots, (S_N, \sigma_N)\}$, where σ is its standard deviation. Every algorithm is evaluated with the same dataset to understand the behavior of each scheduler for the same scenario. There are two types of exploration spaces (composed of the datasets). On the one hand, a large one with 200 sets, centered around $S_{large} = (\{100, 100\}, 50)$. Therefore, there will be evenly distributed sets between 50 and 150 for transfer time and frequency. This dataset gives a heterogeneous exploration space to have a general evaluation. On the other hand, the so-called *corner cases* are evaluated with four different datasets of ten sets each. They are centered around $S_{cc1} = (\{20, 20\}, 10)$, $S_{cc2} = (\{20, 180\}, 10)$, $S_{cc3} = (\{180, 20\}, 10)$ and $S_{cc4} = (\{180, 180\}, 10)$. These represent short and long transfer times and frequencies in extreme conditions, and as σ is small, these exploration spaces are homogeneous, and focused on small areas around the centers.

The simulation time for the large dataset is 100*us*. The sets are heterogeneous enough, so it is a mix of long and short slack, and it is enough simulation time for a proper evaluation. The simulation time for the corner cases is 500*us* because when either the transfer time of

frequency is large, there is less slack, so datasets with a large number of accelerators are preempted more (mainly the dynamic priority ones), so they need more time to complete their transactions. Hence, to have equal comparisons for all four corner cases, all of them have the same simulation time. These four ones do not require many datasets as their sets are homogeneous due to the small standard deviation. All simulations were performed at 100*Mhz*.

Three different characteristics are evaluated for the proposed schedulers. *Scalability* shows how FFs and LUTs scale up to support large number of accelerators. *Schedulability* explores how many requests are actually granted. *Performance* is measured in multiple metrics, providing each of them with different characteristics of the schedulers.

4.3.1 Scalability

The design of the schedulers is meant to rely only on LUTs and FF. An analysis of how resource utilization scales up is shown in Table 4.1. Figure 4.5 shows that both LUTs and FFs have linear behavior, which is desired for larger designs, so the resource consumption does not explode.

Table 4.2 shows the ratio for resource utilization between the two versions of EDF. It can be seen that there are 5% less FFs for ROEDF but 40% less LUTs *in average*. Hence, there is a tradeoff between resource utilization and latency as ROEDF consumes less, but its response time and lateness (c.f., Section 4.3.3) are higher than for LOEDF.

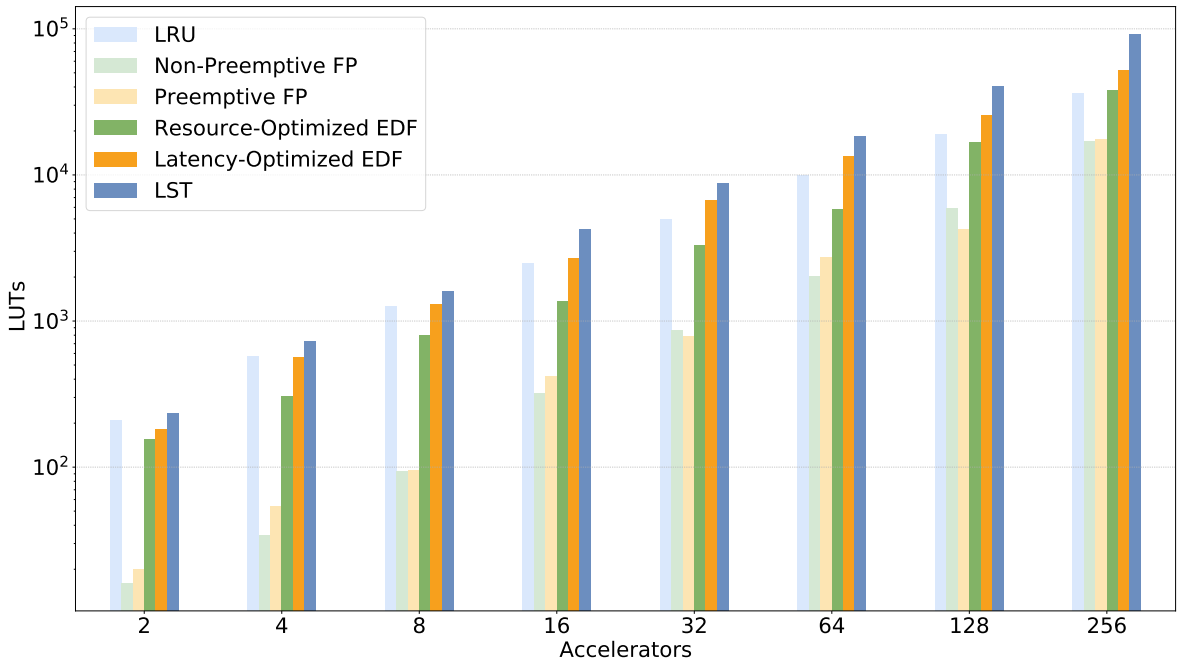
4.3.2 Schedulability

The schedulability is studied to understand the different algorithms' capabilities to schedule tasks (give accelerators the grant). It has a significant impact on the evaluation of the performance done below. It is important then to evaluate the different scenarios by scaling up the design. In order to do so, the characteristics of each accelerator have to have representative values to provide a challenging scenario.

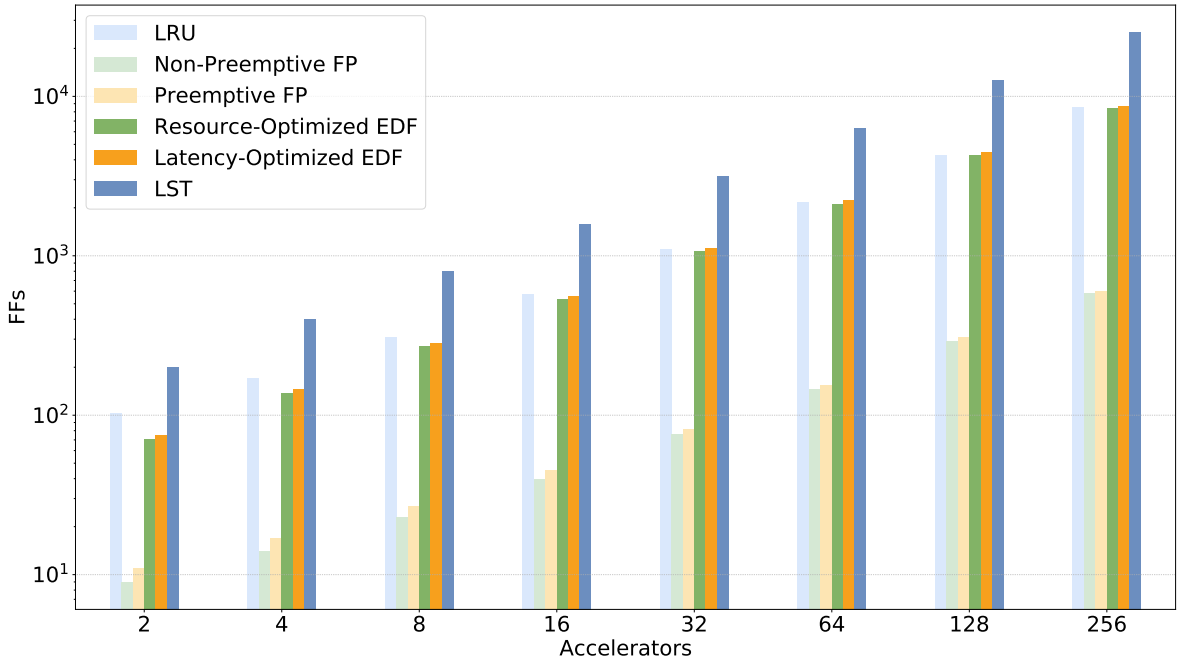
The total number of accelerators that got the grant at least once are shown in Figure 4.6. These numbers can be further analyzed by dividing them by how many of those were able to complete at least one transaction (full bars) and how many did not (striped bars). To further understand the schedulability, Figure 4.7 shows each algorithm's average preemptions per

Table 4.1: Schedulers' resource utilization

Accelerators	LRU		Non-Preemptive FP		Preemptive FP		Resource-Optimized EDF		Latency-Optimized EDF		LST	
	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs
2	209	103	16	9	20	11	156	71	180	75	233	201
4	570	170	34	14	54	17	304	138	565	145	726	399
8	1261	307	94	23	95	27	799	271	1298	285	1593	799
16	2494	578	321	40	421	45	1365	536	2697	561	4246	1582
32	4999	1106	858	76	783	82	3288	1065	6659	1111	8790	3167
64	9971	2172	2030	146	2715	155	5772	2122	13375	2234	18478	6346
128	18826	4312	5931	292	4249	307	16704	4298	25459	4473	40516	12637
256	36273	8521	17023	583	17552	601	38224	8460	52009	8745	92123	25136



(a) Schedulers' LUTs



(b) Schedulers' FFs

Figure 4.5: Schedulers' resource utilization

Table 4.2: Resource-optimized vs. latency-optimized EDF tradeoff

Accelerators	ROEDF/LOEDF	
	LUTs	FFs
2	0.87	0.95
4	0.54	0.95
8	0.62	0.95
16	0.51	0.96
32	0.49	0.96
64	0.43	0.95
128	0.66	0.96
256	0.73	0.97

accelerator (per completed transactions). There is a clear difference of LST to the other dynamic priority algorithms, as this one preempts accelerators at least four times more. The reason is that this algorithm not only considers the time to the deadline but when the request was set (unlike EDF), which has a significant influence on the slack, which translates to more priority updates making it preempt the accelerators more often. These have consequences when many accelerators are in the architecture (128 and 256) that more accelerators get the grant, and not all of them can complete the transactions in the simulation time set for the evaluation. However, a longer simulation time allows more accelerators to complete their transactions. So, it is not a flaw of the scheduler but a restriction on the evaluation methodology. A simulation time of 100*us* provides good results to obtain an appropriate general understanding of the algorithms, as more accelerators complete their transactions. Extending this time did not modify the results shown here; it only improved the schedulability, as expected.

The FP schedulers stand out in Figure 4.6 as they cannot give the grant to many accelerators,

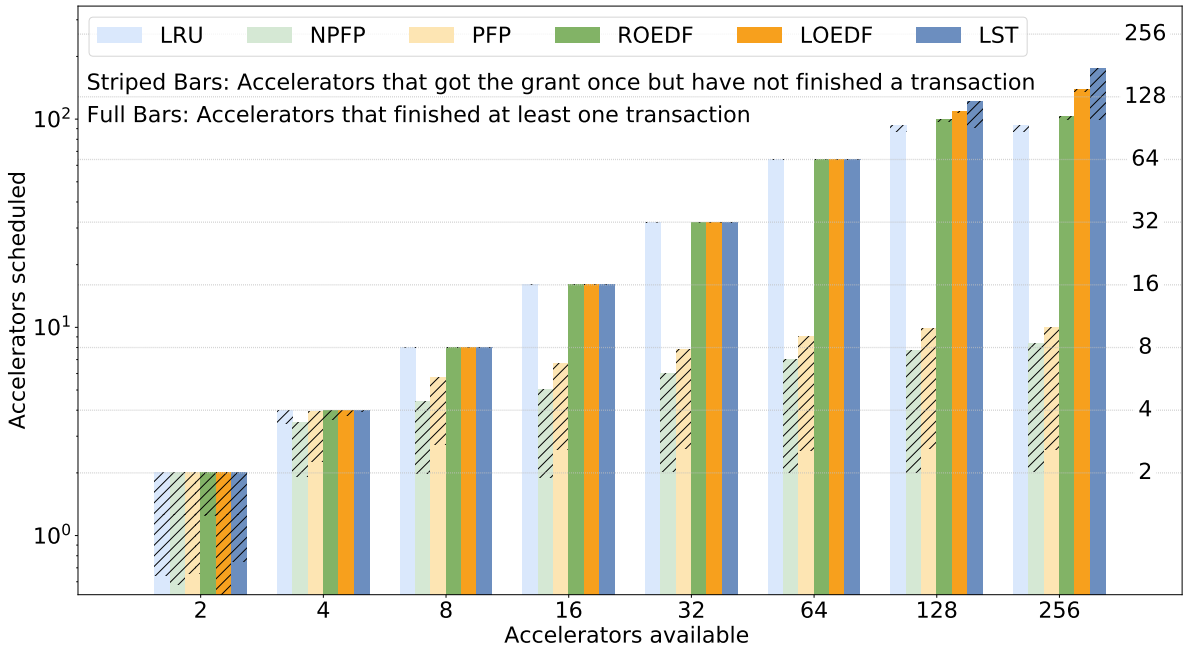


Figure 4.6: Accelerators that finished or got the grant

with a small maximum (around eight) compared to the other algorithms, and just one or two can complete their transactions. This is expected as all accelerators have the same priorities during execution time, and the ones on the top of the priority list will be scheduled regularly. It can be seen in Figure 4.6 the differences between the two non-preemptive algorithms. The counterpart of the limitations of NPFP mentioned before can be seen with the LRU, with completed transactions for almost all accelerators that get the grant. This algorithm

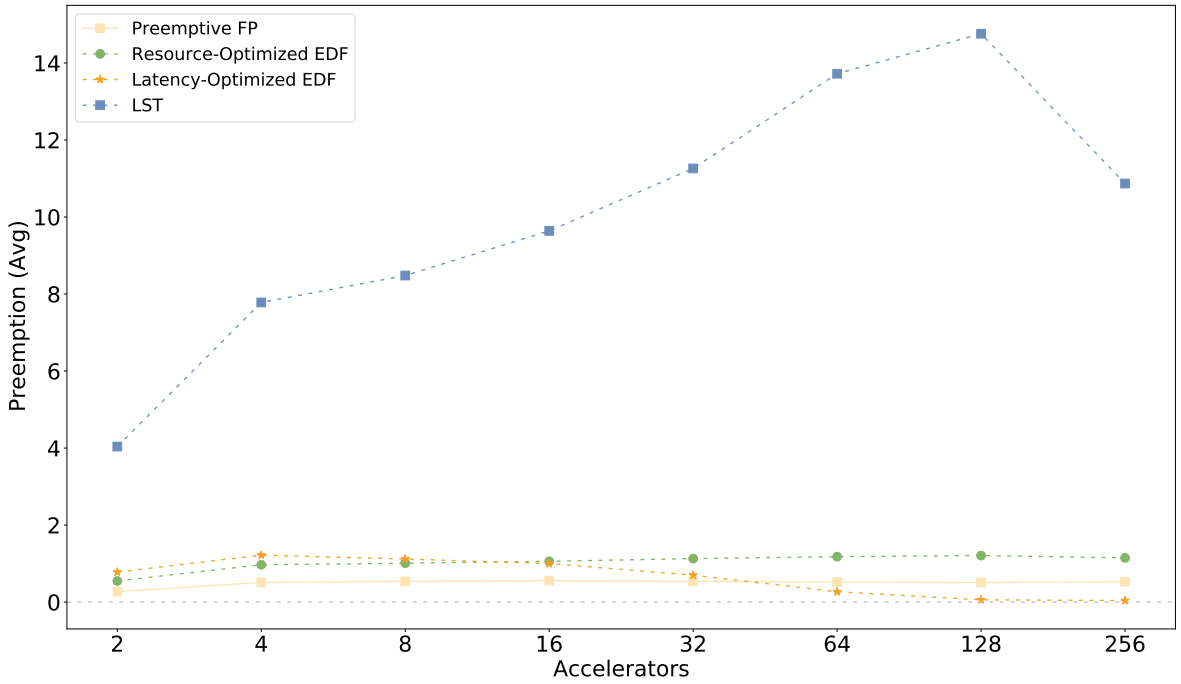


Figure 4.7: Preemptions per accelerator (per completed transaction)

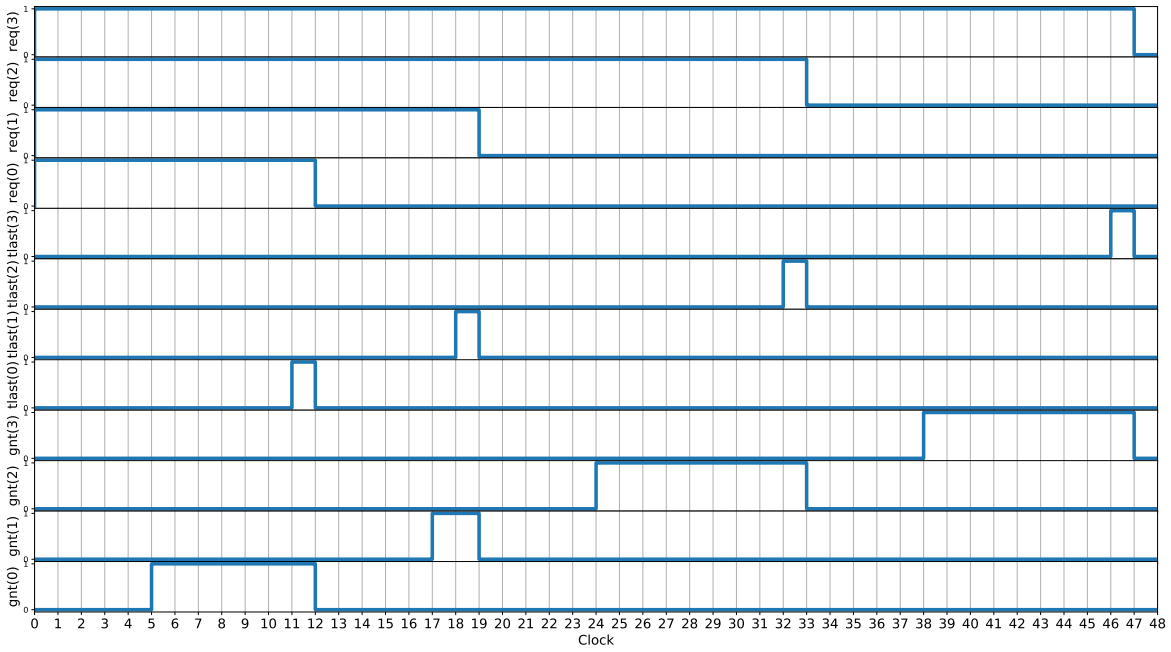


Figure 4.8: LRU example with four accelerators

ensures that all accelerators will get a grand at some point, as shown in Figure 4.8 [1], with the compromise of missing some deadlines. The LRU is a dynamic priority scheduler that ensures each accelerator gets the grant. When an accelerator completes its transaction, the scheduler increases the priority of the remaining accelerators by one. This process continues until all the accelerators are eventually placed at the top of the priority list for scheduling. Even though the priority of all accelerators is increased to reach the top of the list, some may miss their deadlines. However, it is ensured that all accelerators can complete their transactions eventually. Nevertheless, as shown below, this has some drawbacks with its performance.

4.3.3 Performance

The metrics shown below characterized the performance of the different scheduling algorithms.

Average Response Time

The *response time* (r_i) represents how long it takes for an accelerator to get the *grant* since the moment it set the *request*. This can be expressed as shown in Equation (4.3)

$$r_i = g_i - a_i \tag{4.3}$$

where g_i is the time at which the grant was set and a_i is the arrival time of the request. Figure 4.9 shows what the *response time* represents in terms of signals.

The *average response time* (r_{avg}) for n completed transactions is defined by Equation (4.4).

$$r_{avg} = \sum_{i=0}^{n} \frac{r_i}{n} \tag{4.4}$$

Table 4.3 shows the minimum, average and maximum response time measured for all proposed algorithms with the eight different variants of accelerators as inputs. Figure 4.10

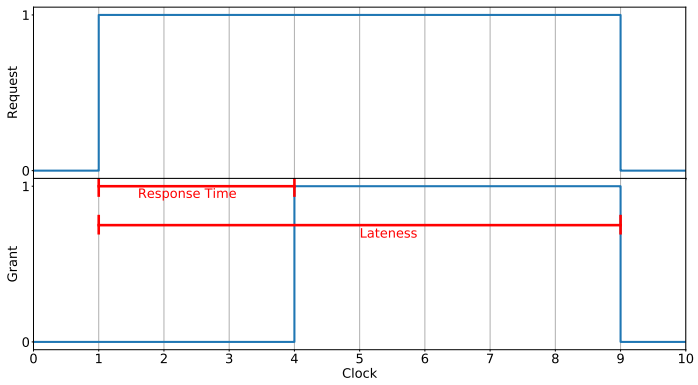


Figure 4.9: Response time and lateness metrics

[1] All signals are obtained from a Vivado simulation and re-plotted with a custom Python script for homogeneous formatting with the rest of this dissertation.

shows the average response time. It can be seen that both FP versions are the ones with the shortest response time, which would lead to thinking this is a good result. However, the schedulability of these two is the worst for all algorithms, as explained before, due to the small number of accelerators scheduled. As expected, LRU is the one with the worst results. This is not an issue as performance is not the main characteristic of this algorithm but ensures accelerator schedulability. LST is the one that shows the best performance with the drawback that it takes a bit longer for all accelerators to complete their transactions. There is a clear difference between both EDF versions, being LOEDF the one with the shorter response time, approaching the same results as LST, but with the tradeoff of more significant resource consumption.

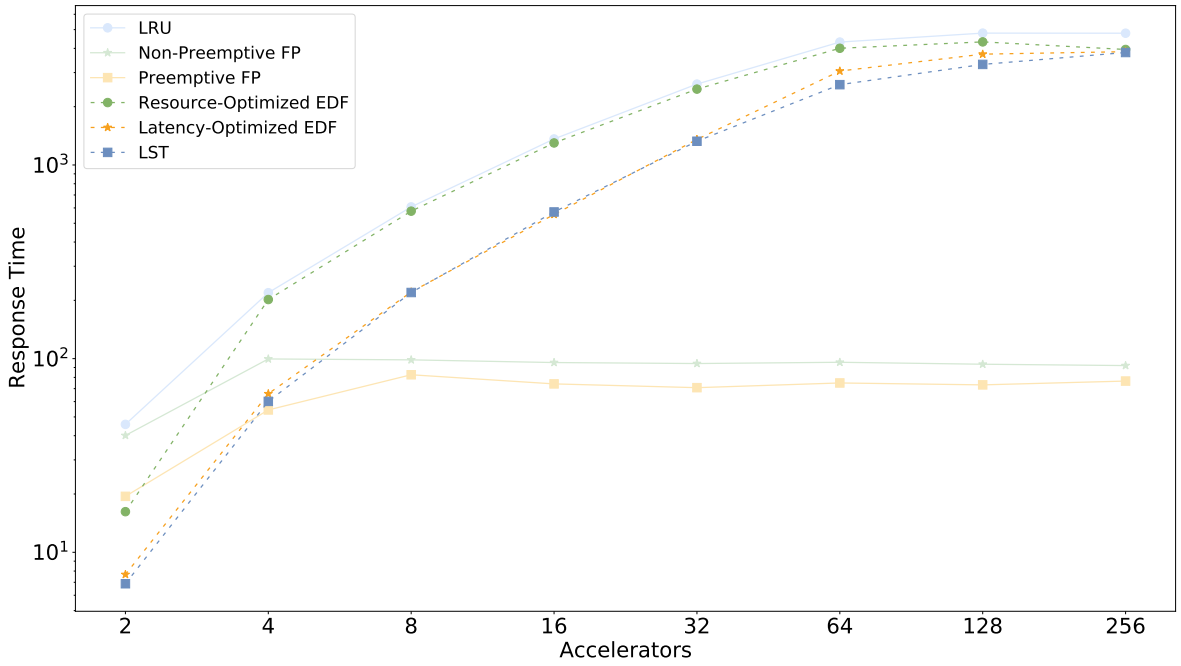


Figure 4.10: Schedulers' average response time

Lateness

Denotes how much later than the deadline the data transmission was completed and is computed as Equation (4.5). A negative *lateness* means the transmission was completed *before* its deadline.

$$L_i = f_i - d_i \tag{4.5}$$

The measured lateness is shown in Table 4.4. Note that positive values (TLAST after implicit deadline) are due to the restricted simulation times. In a real scenario, accelerators would meet their deadlines or complete their transactions. The only requirement to measure the lateness is that accelerators must complete at least one transmission. Similar to the response time, the lateness (Figure 4.11) shows that accelerators with both FP finish the transactions before, but at the expense of not scheduling a large number of them. Also, LRU has the largest lateness. For this metric, both EDF versions outperform LST since the latter one will preempt more accelerators leading them to finish their transactions in a longer time. LOEDF

Table 4.3: Schedulers' response time

Accelerators	LRU			Non-Preemtpive FP			Preemtpive FP			Resources-Optimized EDF			Latency-Optimized EDF			LST		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
2	4.99	45.77	101.78	2.0	40.11	94.98	2.0	19.45	74.5	2.8	16.21	63.68	2.92	7.68	45.7	1.78	6.88	44.37
4	4.12	219.02	316.92	2.0	99.62	225.12	2.16	54.34	770.14	2.84	201.91	299.82	2.88	65.85	300.87	1.94	60.16	303.82
8	3.58	607.5	754.82	2.0	98.55	229.9	2.4	82.55	2662.09	2.8	577.5	759.2	2.69	220.9	729.66	1.99	219.52	708.05
16	2.82	1362.62	1637.6	2.0	95.46	221.46	2.38	74.06	2656.4	2.68	1299.33	1651.47	2.48	555.13	1582.08	1.99	571.94	1526.0
32	2.24	2616.18	3336.02	2.0	94.34	227.8	2.3	70.81	2476.12	2.58	2464.68	3331.88	2.48	1354.82	3219.72	1.99	1326.1	3109.53
64	1.68	4311.82	6812.48	2.0	95.76	227.36	2.23	74.88	2629.71	2.68	4005.35	6730.58	2.52	3055.05	6557.6	1.99	2598.3	6267.49
128	1.5	4795.67	9823.38	2.0	93.57	232.44	2.23	73.2	2643.25	3.08	4320.91	9841.4	2.74	3734.33	9769.36	1.95	3306.09	9606.79
256	1.48	4790.68	9820.51	2.0	92.09	223.32	2.24	76.59	2731.9	3.23	3948.11	9791.94	2.83	3845.78	9830.41	1.71	3808.68	9801.34

Values shown in *number of clock cycles*. 10*ns* used as clock period for simulation.

shows better performance compared to ROEDF, as intended. The reason for this is a shorter latency, which also translates to the smallest lateness for LOEDF among all dynamic priority schedulers.

The maximum lateness (c.f., Figure 4.12) in any given system specification with multiple accelerators can be used to estimate the length of buffers that might be needed to counteract this maximum values.

Communication Channel Utilization

This is a measurement of how much time any of the accelerators get a grant and transmits its data. To be fair with all schedulers, the time for which there are requests is only considered. It is measured following Equation (4.6).

$$U = \sum_{i=0}^{M} \frac{acc_i}{t_{sim}} \tag{4.6}$$

where acc_i stands for the total time accelerator acc_i set its request and was given the grant. t_{sim} stands for the total simulation time in clock cycles.

The communication channel utilization measured is shown in Figure 4.13. Same as the other metrics, LRU is the one that performs the worst due to its design to avoid resource starvation, leading to long response time and lateness, which translates to less channel utilization. Both FP schedulers show a high communication channel utilization, but one has to keep in mind the low number of accelerators that are actually able to finish a transaction. However, it is worth mentioning that as there is no time to update the priority table, this algorithm reacts fast to give the grant to accelerators. In terms of EDF, the resource-optimized version (ROEDF) takes longer to give grants and also preempts the current accelerator holding the grant every time a new request arrives to recalculate the priorities, which translates into lower channel utilization compared to LOEDF. This directly impacts the channel utilization as accelerators

Table 4.4: Schedulers' lateness

Accelerators	LRU			Non-Preemtpive FP			Preemtpive FP			Resources-Optimized EDF			Latency-Optimized EDF			LST		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
2	-129.11	-65.9	-1.81	-132.74	-72.01	-9.41	-116.17	-60.71	200.4	-130.72	-60.01	41.08	-109.27	-71.48	-34.11	-102.85	-59.11	-25.11
4	-121.05	119.12	253.07	-126.15	1.16	129.63	-109.18	52.86	2548.92	-114.18	111.04	272.58	-83.75	94.79	195.04	-54.54	101.77	295.27
8	-124.66	521.31	730.28	-135.39	2.36	158.5	-118.66	74.11	3536.53	-110.52	503.08	716.41	-81.3	462.78	646.34	-48.3	467.87	707.31
16	-130.75	1260.35	1655.91	-139.96	4.64	152.84	-126.16	70.15	3361.17	-107.31	1211.51	1574.44	-82.71	1131.39	1483.5	-51.07	1150.08	1492.51
32	-143.06	2553.77	3537.98	-147.93	4.94	159.66	-131.65	82.14	3974.65	-120.67	2420.03	3315.93	-93.56	2281.35	3202.5	-75.54	2326.48	3176.69
64	-145.22	4183.75	6797.08	-151.81	5.56	146.49	-139.95	64.78	3567.84	-115.38	3855.92	6538.14	-93.53	3677.72	6485.42	-89.28	3967.88	6231.99
128	-145.06	4596.29	9558.06	-157.15	-7.57	126.57	-142.12	77.27	3934.42	-117.76	4124.37	9564.02	-85.95	3599.57	9401.52	-97.32	4256.53	9423.2
256	-152.03	4596.35	9568.82	-162.45	-0.35	146.43	-149.54	72.94	3244.99	-110.16	3847.78	9571.47	-79.53	3717.64	9540.13	-96.74	4078.41	9526.57

Values shown in *number of clock cycles*. 10*ns* used as clock period for simulation. Positive values mean tlast was set after the implicit deadlines (period+length).

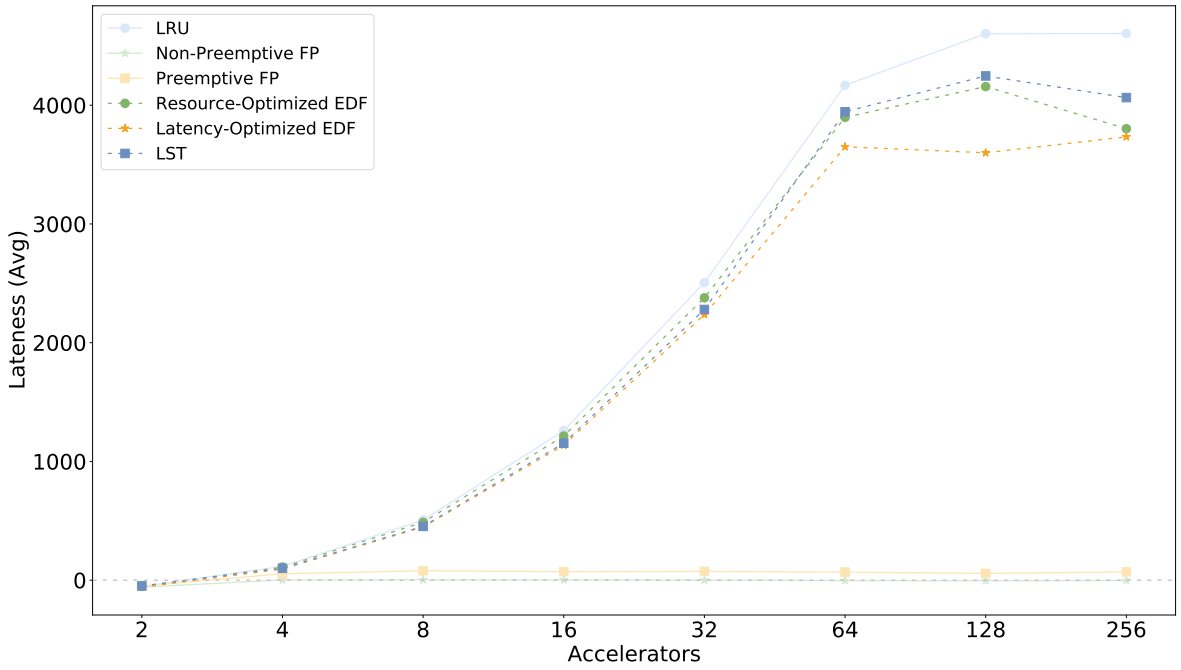


Figure 4.11: Schedulers' average lateness

can stream their data faster (in terms of when each can restart after being preempted). The last point is that as more accelerators get the grant with LST (Figure 4.6), the communication channel utilization is the largest for this algorithm.

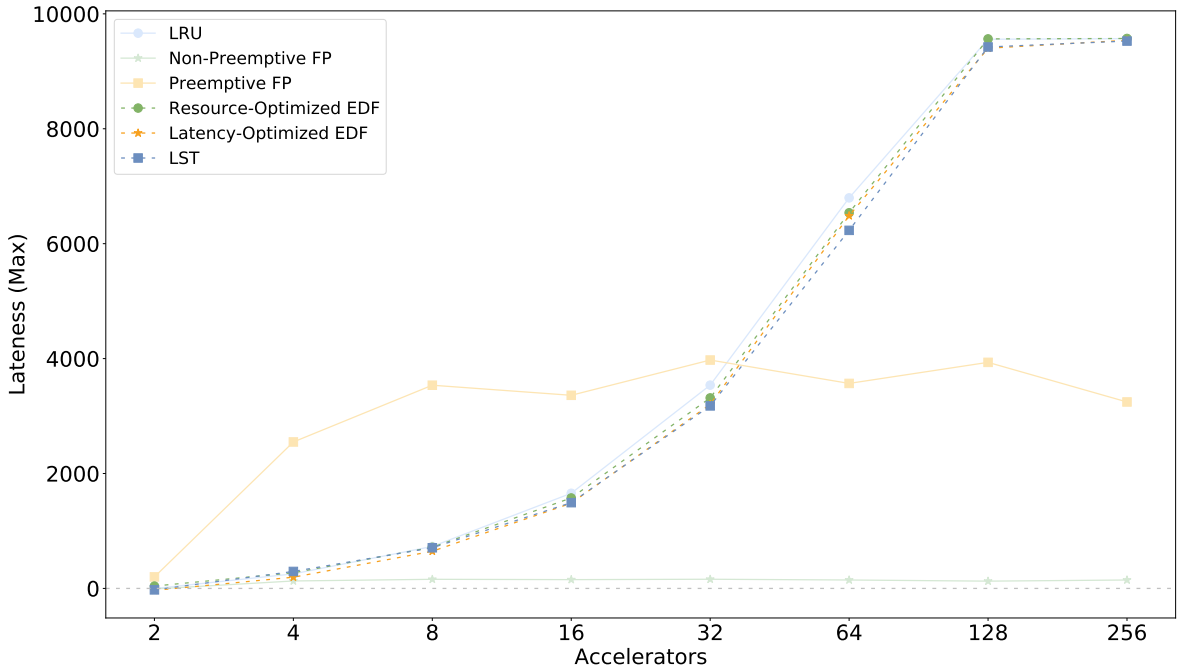


Figure 4.12: Schedulers' maximum lateness

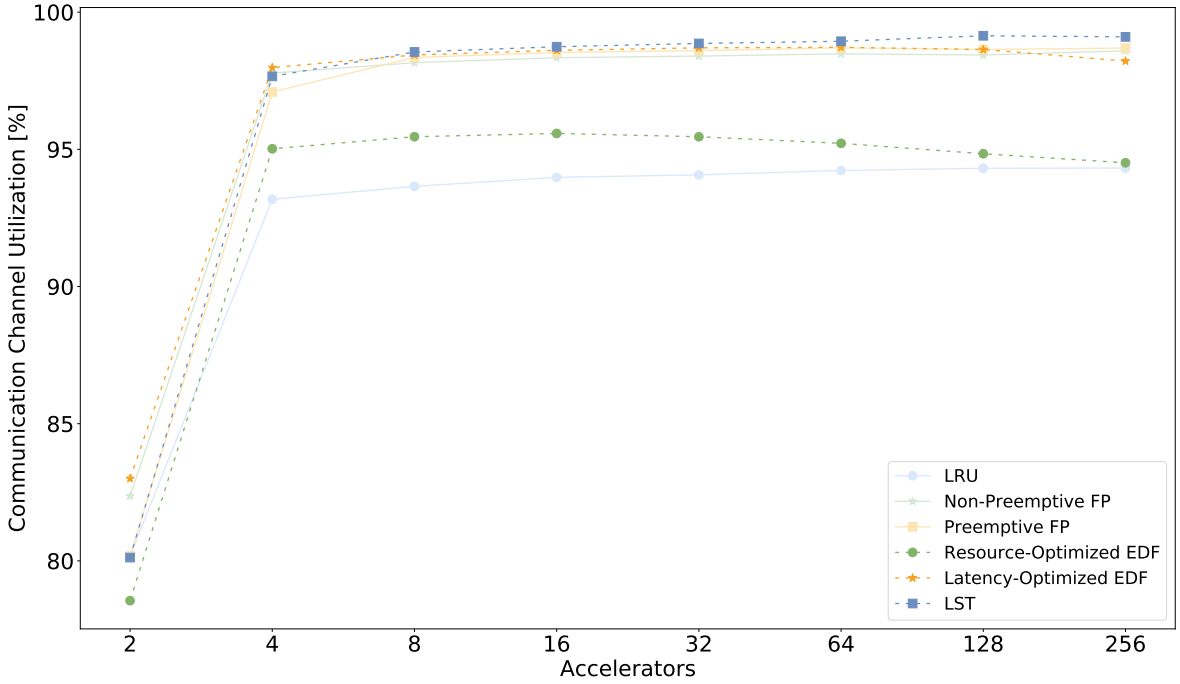


Figure 4.13: Communication channel utilization

4.3.4 Corner Cases

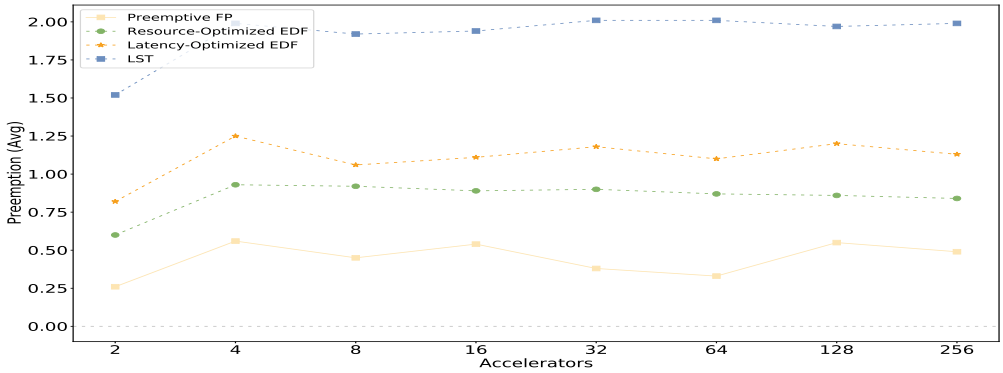
As previously mentioned, four cases with different transfer times and frequencies were evaluated to understand the behavior of the schedulers in these areas of the exploration space. The metrics used previously are also used here to understand their behavior.

Schedulability

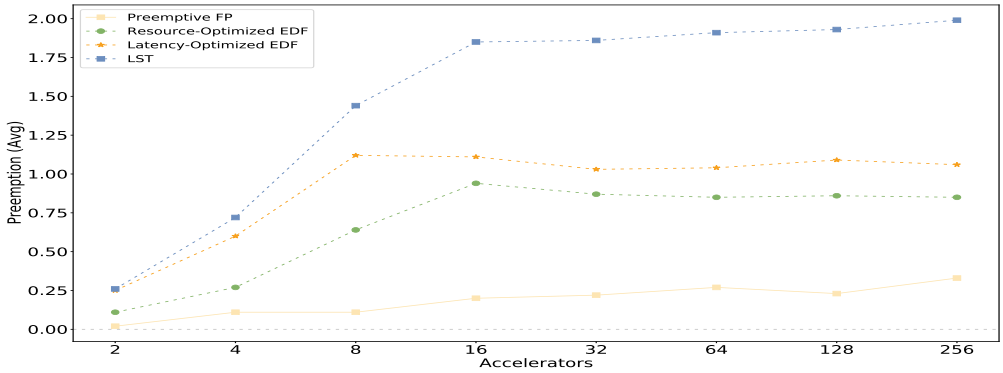
In these four cases, the FP algorithms only schedules a low number of accelerators as before. However, *all accelerators* for the dynamic priority schedulers finished a transaction at least once. The average preemptions per accelerator are impacted by the different transfer times and frequencies, as shown in Figure 4.14. In all cases, LST continues to be the algorithm that preempts most of the accelerators, and the preemptions increase significantly with more significant transfer times, regardless of their frequency. This is clear because each accelerator requires to have the grant for more time to finish a transaction which causes more preemptions. Moreover, these four datasets have a small σ. Therefore, the possibility for *laxity ties* (two or more accelerators with the same priority constantly preempting each) is high.

Performance

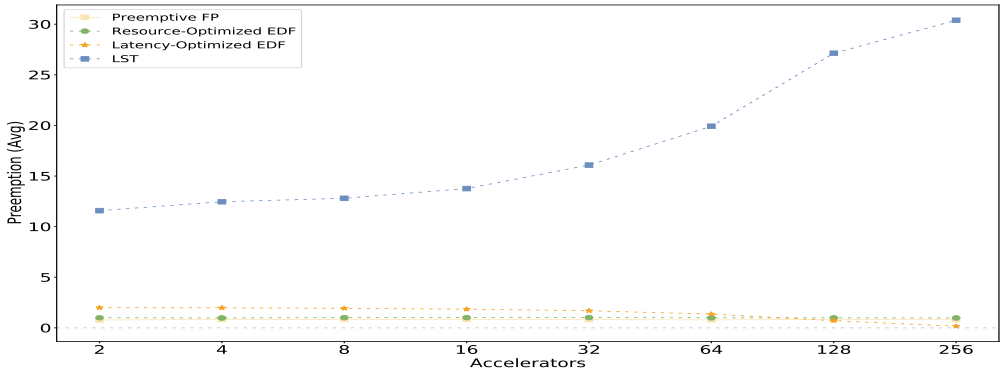
The transfer time of the accelerators affects the response time, increasing it with higher values, as shown in Figure 4.15. It is possible to see that the response time increases by one order of magnitude in the cases with the largest transfer time. Previously, LOEDF and LST



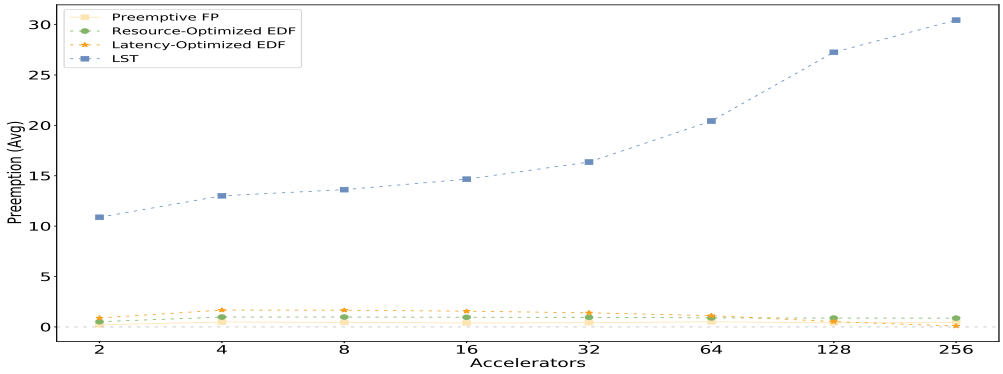
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170



(c) Transfer Time=170 - Frequency=20



(d) Transfer Time=170 - Frequency=170

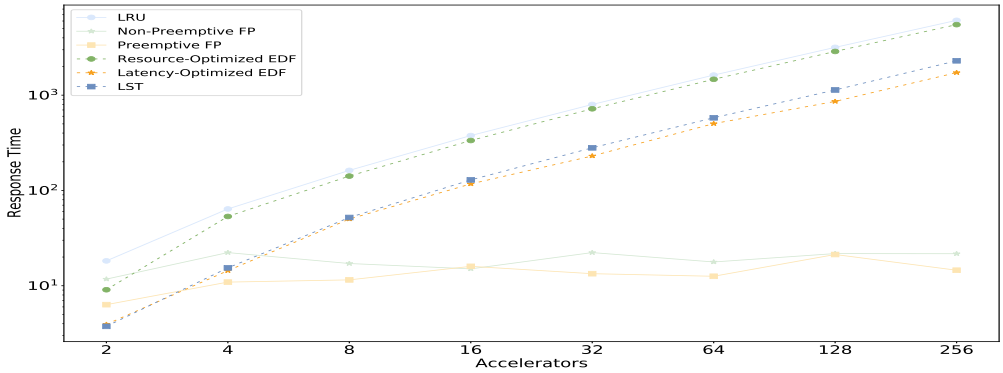
Figure 4.14: Schedulers' corner cases: Average preemption per algorithm

had similar performance. Here, for short transfer times, it is actually LOEDF with a shorter response time (as opposed to LST in Figure 4.10), same as for long transfer times but up to a certain number of accelerators. When more than 64 are present, LST has a lower response time, making it a better candidate for this situation. The lateness is affected by the shortest period, as it takes longer for the accelerators to complete their transactions, either when their frequency is short or long (Figure 4.16a and Figure 4.16b). Figure 4.16c and Figure 4.16d depict the worst-case scenario when the transfer time is the longest, meaning that it takes significantly more time (one order of magnitude) to finish. Note how LST diverges from the other accelerators after 128 accelerators due to the significant increase of preemptions at this point. As for LRU, it is the algorithm with the worst performance for these corner cases because its goal is to ensure that all accelerators can finish their transactions at least once.

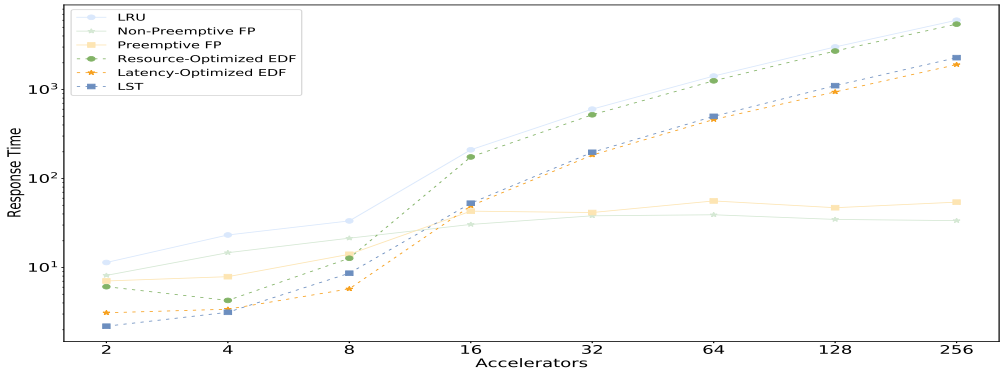
The longer the transfer time, the higher the channel utilization (Figure 4.17). This is particularly clear for LRU, with an increase of 20% (Figure 4.17a and Figure 4.17b vs. Figure 4.17c and Figure 4.17). The frequency decreases mainly the channel utilization with high values, and when there are few accelerators because there are extended periods without any requests, reducing the channel utilization. However, when many accelerators are involved, there will almost always be at least one requesting the grant, even though the channel utilization never reaches 100%.

4.3.5 Combined Schedulers

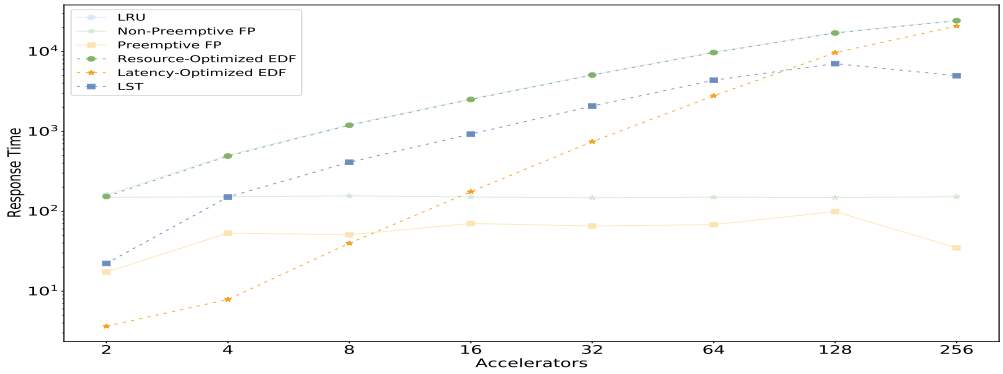
The possibility of improving the schedulability can be achieved by smartly combining different schedulers. For this, a baseline with a dataset of 256 accelerators was used to evaluate how many get the grant with LOEDF and LST as these two showed to be the most promising ones, and the question is whether the results shown previously can be improved. Two different cases are studied. The first consists of splitting the accelerators into smaller datasets, in this case, dividing one large scheduler with 256 accelerators into *two of the same algorithm* but with 128 accelerators each. The second one also splits into a smaller number of accelerators per scheduler, but with *two different algorithms*. All these require a third scheduler also to manage the new smaller ones. *LRU* is chosen for this study as it ensures that all requests get a grant. The results are shown in Figure 4.18. Splitting them does not increase the number of accelerators that got the grant for LOEDF but increases 1.16x for LST. Combining schedulers resulted better for LOEDF (1.29x) as it was done with LST, which showed better schedulability. However, in the LST case, combining it with LOEDF was, in fact, detrimental. Note that every combination of schedulers is possible. However, it does not guarantee improved schedulability as the decision on which schedulers to pick for the best result should be done following a similar design exploration as shown previously for the evaluation of each algorithm. The resource consumption of these combinations is shown in Table 4.5.



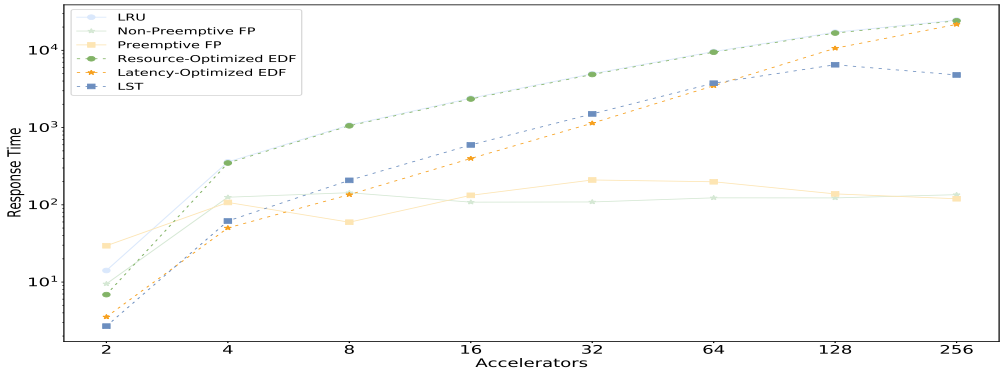
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170

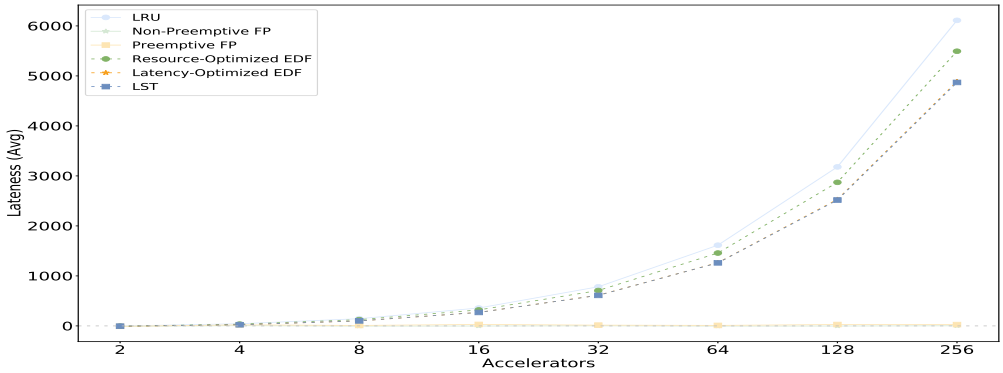


(c) Transfer Time=170 - Frequency=20

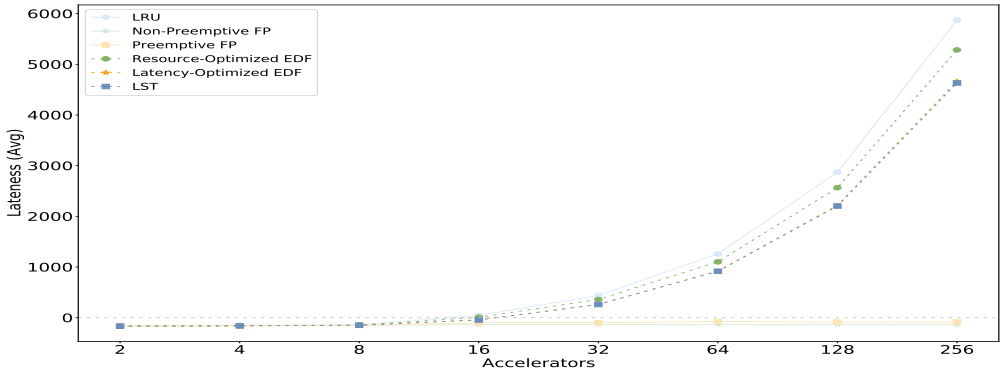


(d) Transfer Time=170 - Frequency=170

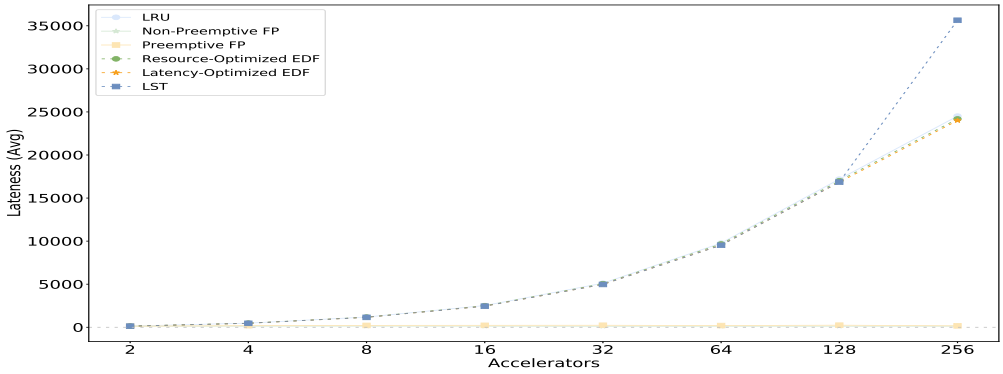
Figure 4.15: Schedulers' corner cases: Response time



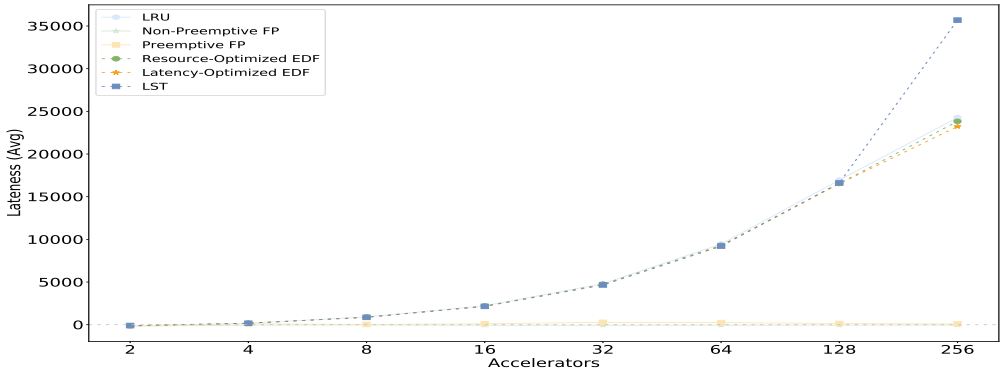
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170

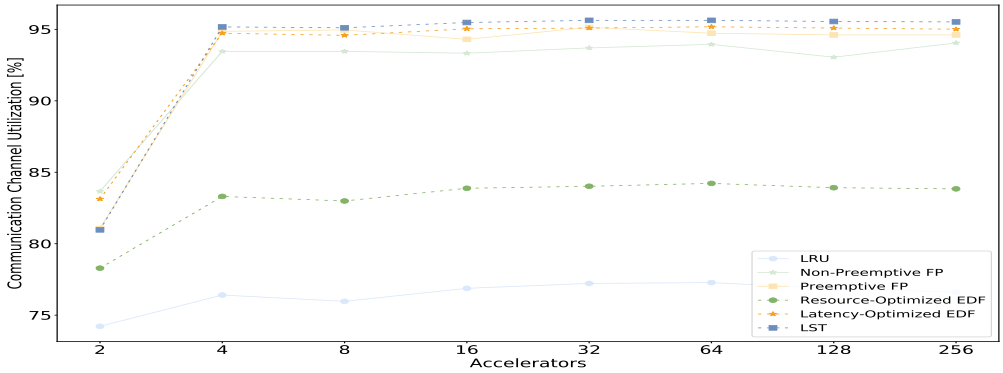


(c) Transfer Time=170 - Frequency=20

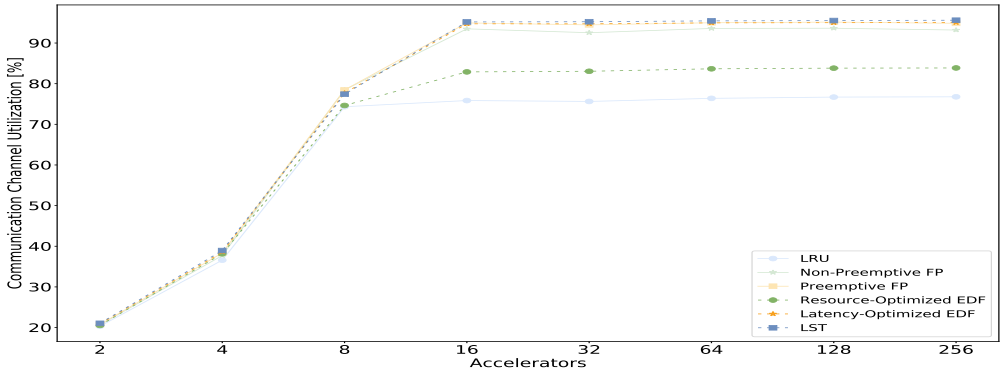


(d) Transfer Time=170 - Frequency=170

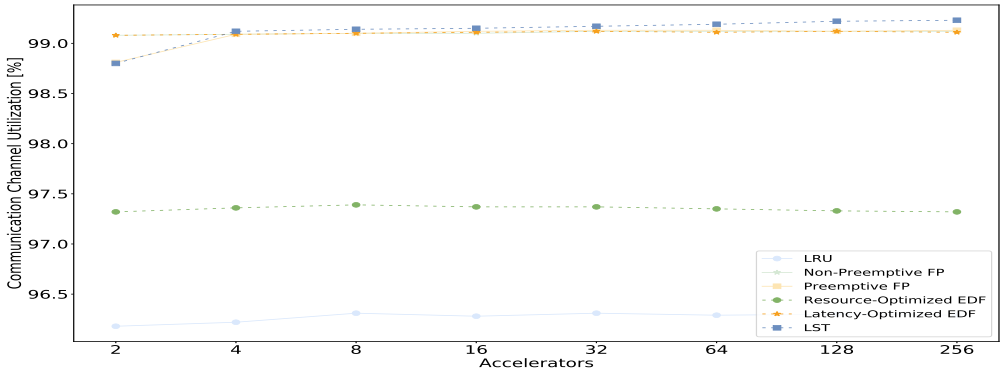
Figure 4.16: Schedulers' corner cases: Lateness



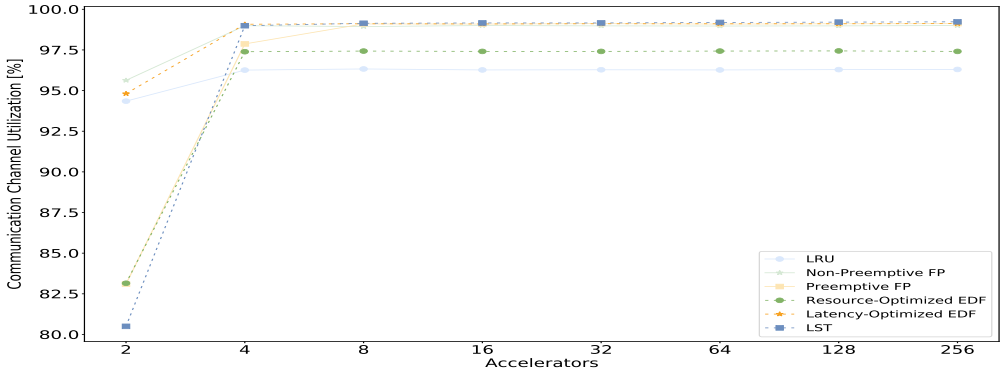
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170



(c) Transfer Time=170 - Frequency=20



(d) Transfer Time=170 - Frequency=170

Figure 4.17: Schedulers' corner cases: Channel utilization

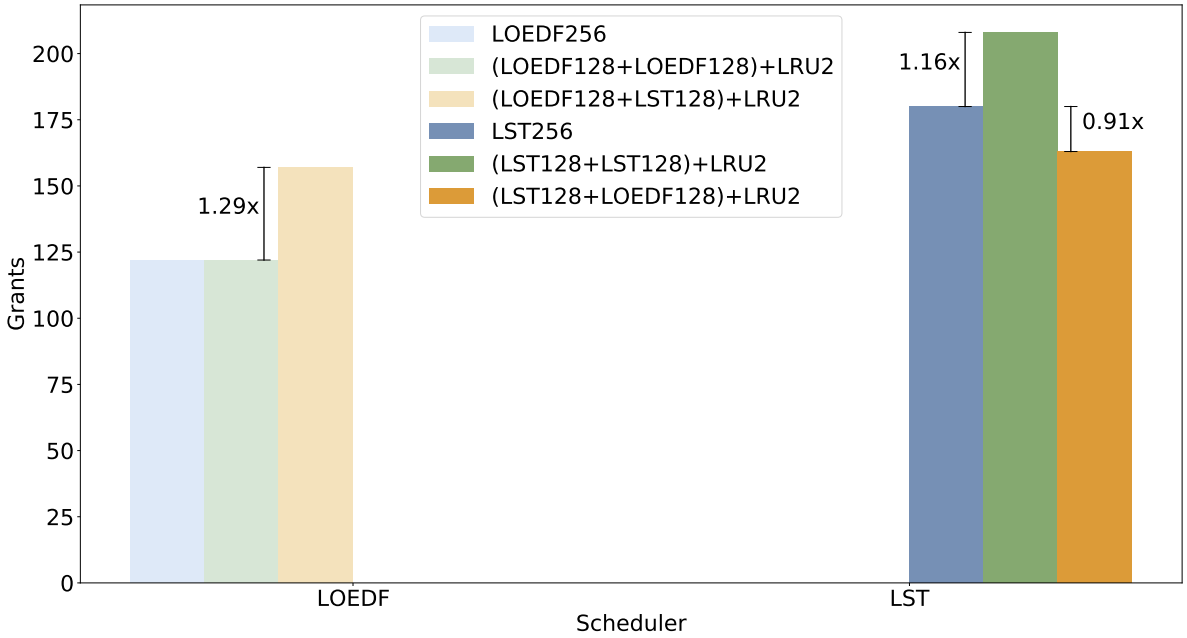


Figure 4.18: Combined schedulers

4.4 Schedulers Comparison

An overview of the proposed schedulers described above is shown in Table 4.6. It describes the features to consider when choosing the best scheduler for a given application, also considering some requirements.

For example, in a situation with limited resources, one could look at the table to check which scheduler offers the minimum usage of LUTs and FFs. However, the table also shows which are the algorithms that present a high number of accelerators that completed their transactions (which are different from the ones that consume fewer resources). So, in this case, NPFP and PFP schedulers are the ones that consume fewer resources, but they do not ensure that all accelerators will finish their transactions, which implies a trade-off that will depend on the application. If there are not many accelerators in the given application, then the FP schedulers will meet the requirements.

Another example could be the case where the application presents between 32 and 64 accelerators, and they must finish their transactions. Table 4.6 shows that LOEDF and LST

Table 4.5: Combined schedulers' resource utilization

Schedulers	LUTs	FFs
LOEDF256	52009	8745
(LOEDF128+LOEDF128)+LRU2	64142	17109
(LOEDF128+LST128)+LRU2	78175	25277
LST256	92123	25136
(LST128+LST128)+LRU2	86673	33515
(LST128+LOEDF128)+LRU2	75578	25315

Table 4.6: Schedulers comparison

Accelerators	Minimum LUTs	Minimum FFs	Most Accelerators with completed transactions	Least Preemptions	Minimum Response Time	Minimum Lateness	Maximum Communication Channel Utilization
2	NPFP	NPFP	ROEDF	PFP	LST	NPFP	LOEDF
4	NPFP	NPFP	LST	PFP	PFP	NPFP	LOEDF
8	NPFP	NPFP	LRU, ROEDF, LOEDF, LST	PFP	PFP	NPFP	LST
16	NPFP	NPFP	LRU, ROEDF, LOEDF, LST	PFP	PFP	NPFP	LST
32	PFP	NPFP	LRU, ROEDF, LOEDF, LST	PFP	PFP	NPFP	LST
64	NPFP	NPFP	LOEDF, LST	LOEDF	PFP	NPFP	LST
128	PFP	NPFP	LOEDF	LOEDF	PFP	NPFP	LST
256	NPFP	NPFP	LOEDF	LOEDF	PFP	NPFP	LST

meet this requirement. However, the table also shows that, on the one hand, LOEDF preempts the accelerators the least, which could be better for the given application. On the other hand, LST has a higher communication channel utilization, which might be a requirement to consider when selecting the most suitable scheduler.

4.5 Summary

The base architecture is meant to deal with software and hardware components, meaning that data is exchanged between two different types of systems. Both systems communicate over a shared resource, namely DMA. As multiple components can be on each of them, the access to the DMA, in both directions, must be arbitrated. Therefore, a hybrid software/hardware scheduler is needed. This chapter presents six scheduling algorithms to be part of the `Manager` to manage the transactions between PS and PL. It is a hybrid scheduler to provide fair access to the shared resource from a software component to its hardware counterpart and vice-versa. Several algorithms are proposed in this chapter to be versatile and have multiple options for different applications. Same as for the other components of the base architecture, they are vendor-independent as they are based on the VHDL-93 standard. The implementation methodology for all algorithms focuses on scalability and adaptability, easing their code generation tailored for different use cases.

The evaluation focuses on the proposed scheduling algorithms, studying their scalability, schedulability, and performance. These metrics help understand the behavior for different scenarios. A comparison of all schedulers derived from this evaluation helps to decide which algorithm will be the most fitted for a given application. The evaluation focuses on the hardware components as there are more demanding requirements, mainly in terms of resources (i.e., LUTs, FFs). The scalability and schedulability are mainly independent of where it is implemented (software or hardware).

5 Generation of Hardware Interfaces Compatible with Robotics based on Specifications

Increasingly complex robotic platforms incorporate heterogeneous sensors and actuators. They are usually coupled with embedded computers but rely on software solutions not entirely suited for processing a large amount of data concurrently and fast enough to keep real-time constraints. FPGAs are ideal candidates to enhance those systems' computing capabilities while still being programmable. They are used in a wide variety of applications due to their intrinsic parallelism capabilities for algorithms, their flexibility, and energy efficiency. However, they impose some challenges to be combined with software solutions. It is cumbersome to manually incorporate them into new or existing systems because providing accelerators with a specific integration capability limits their applicability. The goal is to seamlessly replace software components with FPGA-based ones while retaining the same communication interface. Therefore, designing scalable and reusable interfaces between these two is desired to achieve good synergy between FPGAs and software systems. This chapter presents an approach to generating hardware interfaces for accelerators compatible with robotics based on message specifications. A model-based toolchain automatically generates the necessary hardware components (VHDL modules) from existing message specifications to exchange data with the accelerators. Instead of writing several hundred lines of VHDL, a dozen input specification lines are sufficient with the approach presented in this work. The results are validated by evaluating all message specifications included in the latest ROS versions. The 3102 messages from ROS1 and 1346 messages from ROS2 evaluated show the robustness of the approach's capabilities to support arbitrarily large ROS messages types, multiple data types, and nested messages. Moreover, the approach facilitates the extension from ROS1 to provide support for ROS2 easily. Finally, two use cases are shown to prove the approach's feasibility in real applications. The first one incorporates a hardware accelerator for image processing obtained by HLS into an existing software architecture. The second one consists of a fully FPGA-based mobile platform with ROS features incorporated.

5.1 Challenges and Goals

Despite all the advantages of FPGAs, the robotics community has not fully included them so far as part of their systems for several reasons. First, designing FPGA-based solutions requires hardware knowledge and longer development times than software solutions. Second, porting

a robotics application (or parts of it) from software to an accelerator requires adequate interfaces between software and FPGAs. Consequently, there is a need to investigate new approaches to take advantage of the concurrent processing capabilities of FPGAs and to easily incorporate them into heterogeneous distributed systems to enhance their computational power. These new approaches would ease their utilization by other fields, such as robotics, and improve their processing capabilities. However, these impose the following challenges:

- Challenge 5.1: `Interface Compliance`: Hardware accelerators need to comply with available interface specifications from middlewares and frameworks such as ROS, so they can communicate with other components, either in software or other accelerators.
- Challenge 5.2: `Adaptivity`: Provide flexibility to be extensible for new features, hardware components (e.g., sensors and actuators), and middlewares.
- Challenge 5.3: `Complex Architectures`: Manage multiple accelerators and their communication with other components from the distributed system.

A *holistic approach* requires considering all three of them. Challenge 5.1: Interface Compliance and Challenge 5.1: Adaptivity are the aims of this chapter, and Challenge 5.1: Complex Architectures is addressed in Chapter 6.

Needless to say, a workflow to integrate these three challenges must be available to include FPGAs in robotic applications without increasing the complexity of the traditional robotics workflow design. Therefore, the *goals* shown in this chapter are a flexible model-based workflow to generate hardware interfaces (`message-dependent` components) for accelerators based on robotics message specifications with:

- an open-source toolchain providing code and configurations to create hardware components automatically.
- seamless integration of the autogenerated parts into a hardware architecture capable of handling and interfacing multiple accelerators.

The extension of the toolchain to generate and automatically deploy hardware architectures is further detailed in Chapter 6, as mentioned before.

5.2 FPGA Interfaces for Robotics Middlewares (FIRM) Tool

As illustrated in Chapter 3, the `message-dependent` hardware components have to be created since accelerators have to be aware of the *semantics* of the data structure they process. For ROS components, this is done using software client libraries, which provide serialization and deserialization functions to translate the received data into the concepts of the respective programming language the component is written in. For example, the *gencpp* tool[1] creates the header files containing the data structures and methods required to process messages in C++.

In this work, the goal is to construct a similar tool to create the required `message-dependent` components. These are VHDL files to convert ROS messages from and to an AXIS frame. An example is shown in Figure 5.1. It can be seen the equivalence between the message specification (Figure 5.1a) and the resulting hardware entity (Figure 5.1b) in VHDL. The

[1] `http://wiki.ros.org/gencpp`

resulting AXIS frame is shown in Figure 5.1c, where each of the variables are split in bytes (TDATA) and streamed in the order they are defined in the message specification. However, due to the much lower level of abstraction VHDL provides compared to languages such as C++ and Python, this process is much more complicated, mainly because data type sizes are relevant, and there are no built-in language mechanisms to support custom or composite data types. Thus, it is beneficial to describe the properties of both the ROS message formats and the resulting AXIS explicitly using *models* rather than encoding them implicitly in simple scripts.

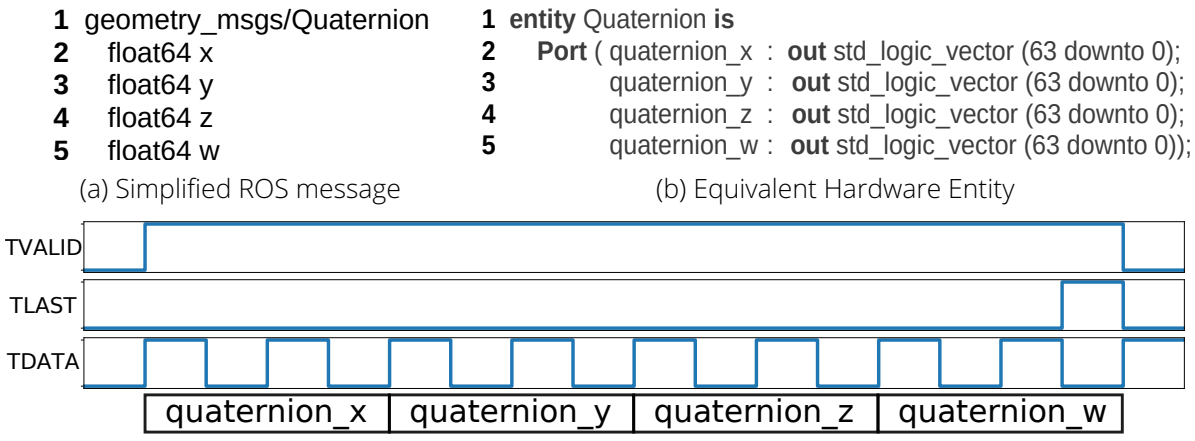
The use of models also allows the use of both an explicit and declarative formulation of the required analysis of the ROS message data structures. This, in turn, enables explicit and declarative transformations of the ROS data structures into the desired target format. Additionally, both the message format and the resulting VHDL code are not fixed and can be adapted to newer requirements. For example, the introduction of ROS2 has already introduced new features into the message definition. Additionally, the backends might change. Not only does the ROS serialization differ between ROS1 and ROS2 (in which furthermore different middlewares and thus serializations can be used), but also VHDL might require different revisions when using other synthesizers or hardware from different vendors.

Therefore, a flexible and extensible *model-driven toolchain* to generate hardware interfaces for ROS components is proposed. All the details and design decisions are shown below.

5.2.1 A Model-Driven Toolchain

MDE [28] describes the technique to use a staged model transformation process. Each additional information added to the final model is used to generate the desired code artifacts, which in this case is a set of VHDL components. The remainder of this section explains the proposed process and highlights the benefits of MDE and model-based code generation.

Figure 5.2 gives an overview of the code generation workflow, centering around the model-driven process implemented in the tool called FPGA Interfaces for Robotics Middlewares (FIRM).



(a) Simplified ROS message

(b) Equivalent Hardware Entity

(c) Resulting AXIS Frame

Figure 5.1: ROS message and hardware equivalence

Using a configuration file and a set of ROS message specifications, a *model-driven code generation* tool constructs the VHDL `message-dependent` components described in the previous chapter. Within the tool, a parser, a sequence of model transformations, a model-to-text generator, and finally, a template engine are used.

The primary input is a single configuration file. An example of such a file is shown in Listing 5.1. Besides some configuration options for the hardware platform (Lines 1 to 5), this configuration contains a list of the ROS messages that are published and subscribed to and the names of all interfaces for publishers and subscribers (accelerators) (Lines 6 to 12). This configuration is parsed into a *configuration model*, which is used to generate both the `accelerator-related`

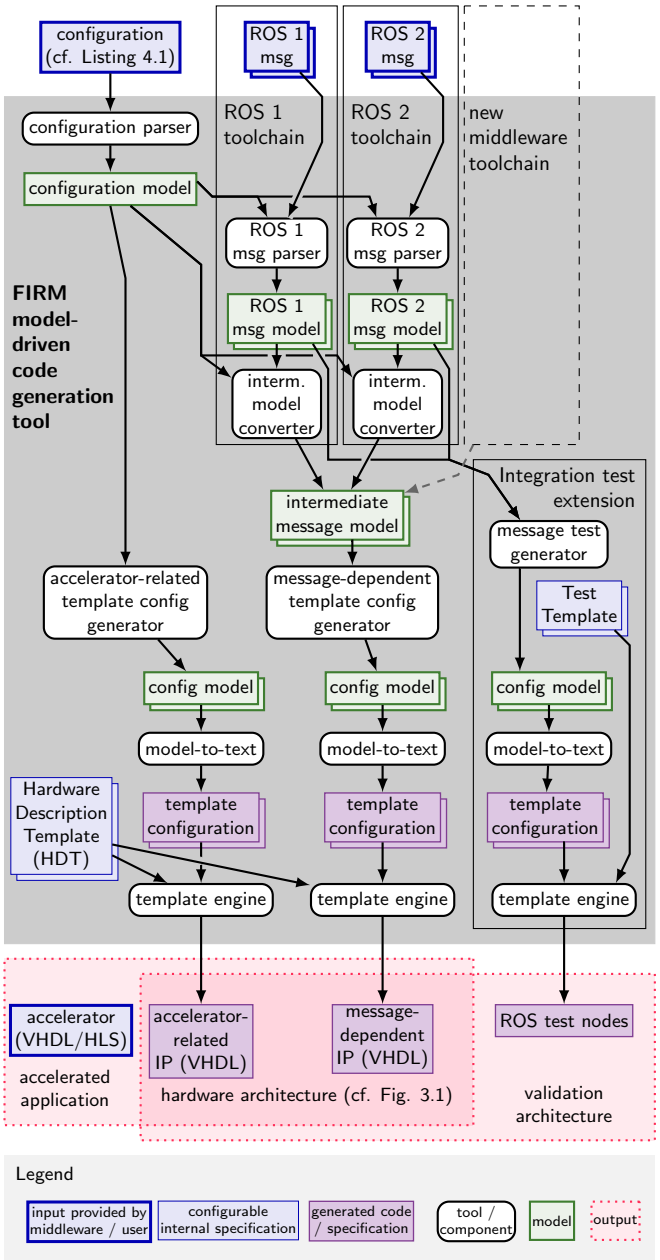


Figure 5.2: Workflow to generate hardware architectures

and the `message-dependent` components of the hardware architecture. Besides the actual accelerator, this is the only artifact that has to be provided for every use case.

While parts of the process depend on whether ROS or ROS2 is used, this is not resembled in the configuration file since it can be deduced from the context in which the tool is run, which adds portability. The generated components require information about the structure of the ROS messages, which serve as interfaces to the accelerator. Therefore, a dedicated *ROS message parser* is used to retrieve the message specifications from ROS and obtain *ROS message models* (see the meta-model in Figure 5.3a). Note that Figure 5.2 shows the possibility to extend FIRM with other middlewares besides ROS and ROS2 following the same approach described here.

Analysis has to be performed in preparation for the interface generation (i.e., to compute the relative positions of its elements) required in the `message-dependent` IP cores, using these models as well as the configuration model. Because this analysis does not have to be aware of all details contained in the ROS `message` format and should ideally be reusable for different message specifications, the ROS `message` model is then transformed into an *intermediate message model*, containing all information to generate the `message-dependent` components (a meta-model is shown in Figure 5.3b). The code for the `accelerator-related` components can be generated using only this intermediate model and the configuration, decoupling the code generation from the message specification.

A logic-less (i.e., containing no complex template expansion logic) template engine is used to separate the resulting logical structure of the code from the concrete syntax. The template engine is configured by files produced by generator components and model-to-text transformations for the `accelerator-related` and `message-dependent` parts. Additionally, the template engines require templates for the desired artifacts (i.e., VHDL files and scripts for the FPGA-related toolchain). Note that while the toolchain generates the template configurations, the templates must be defined manually *once*. Still, one set of templates can be used to generate multiple architectures using *any kind and combination* of messages.

The major conceptual and technical design decisions of the workflow and their relation to the challenges presented in Section 5.1 are described next.

```
1  project:
2    name: Sobel Filter
3    platform:
4      board: zcu104
5      FPGA: xczu7ev-ffvc1156-2-e
6  messages:
7    - type: ROS                # supported middleware
8      name: sensor_msgs/Image  # message specification
9      subscribers:
10       - AXIS2Image           # names for interfaces
11     publishers:              # to midleware used by
12       - Image2AXIS           # accelerators
```

Listing 5.1: Configuration file for an image processing use case

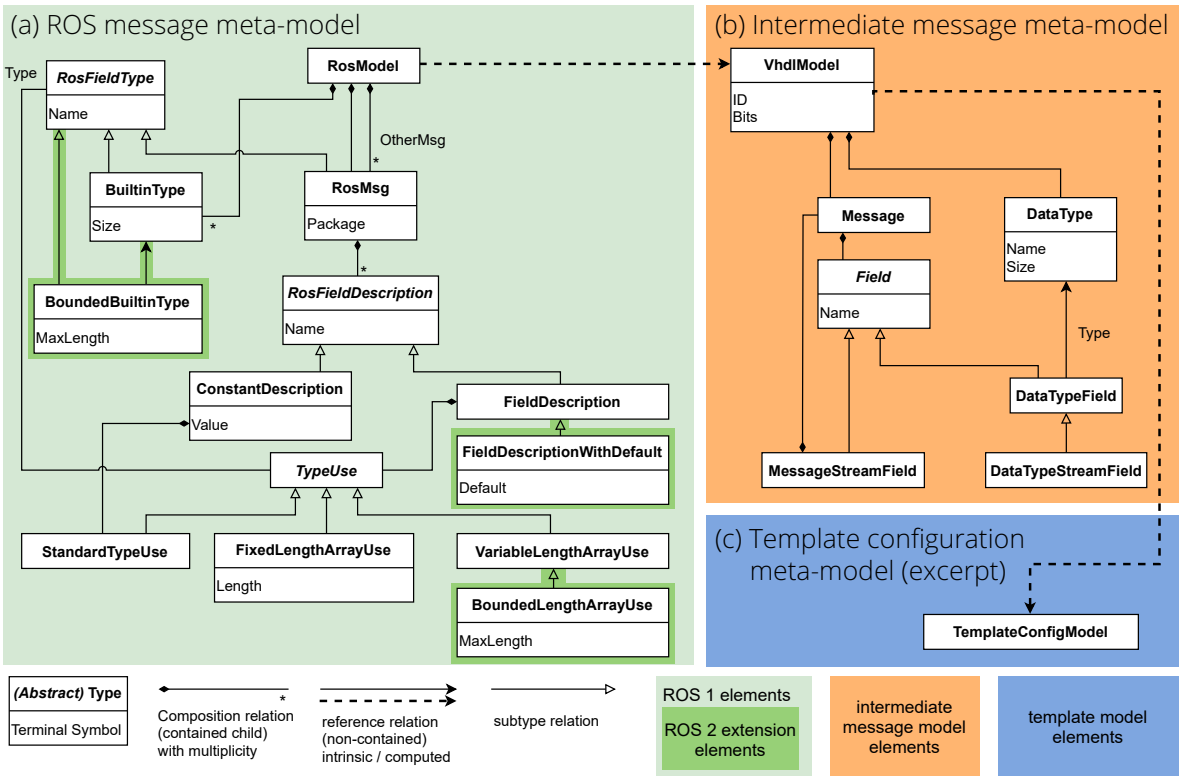


Figure 5.3: Meta-models in the FIRM tool.

5.2.2 Characteristics of the Model-Driven Toolchain

The toolchain is modeled using RAGs [33] and implemented using the RAG system *JastAdd* [36]. Grammars specify a language using tokens (also known as terminal symbols) and non-terminal types with production rules defining the types and order of their contained elements. A sequence of tokens is an element of the grammar if a derivation tree can be found that constructs it using the production rules; this tree is also known as the AST. Attribute grammars are a suitable modeling approach since they provide integrated declarative static analysis by adding semantic-defining attributes to non-terminal nodes in the AST, which are formally specified using equations [116]. In the *JastAdd* system, these equations are defined in a Java-based DSL and thus offer similar features as Java methods. Furthermore, attribute grammars and RAGs were developed specifically for the construction of compilers [117, 118, 119], which the approach presented here classifies for as well because there is a transformation from a source language (a configuration file and a message specification) into a target language (VHDL). The general idea for using attributes in FIRM is to generate tailored code for specific messages. In order to do this, supporting attributes are used to compute, among others, names, types, sizes, and positions of data fields in the AXIS frame. This is a non-trivial task because of message nesting, unconstrained array types, and built-in data types that require further conversions. Furthermore, the model transformation and code generation steps are also performed using *higher-order* attributes [120], computing entire derived models rather than simple properties.

The *relational RAG* extension [121] is employed to be able to handle (graph-shaped rather than tree-shaped) conceptual models such as the configuration and message models described

previously more conveniently. *Relational RAG* adds a secondary graph structure to the AST, thus allowing more concise attribute specifications when dealing with nested message types and field types.

Using RAGs, this approach addresses Challenge 5.1: Adaptivity and Challenge 5.1: Complex Architectures. It follows a well-structured (model-based), formal (grammar- and attribute-based), and concise definition (using attribute equations) of all aspects of the toolchain including configuration, message analysis, and code generation. Next, a closer look at the employed models and attributes to further illustrate the workings and benefits of the chosen approach is presented.

5.2.3 The Models

Figure 5.3 shows a meta-model representation of the employed models. While a grammar-based technique is used, the models are shown in a graphical UML notation for clarity, where grammar production rules are shown using composition edges. Since the grammar specification format of *JastAdd* uses production rule inheritance, this feature is also used here and is interpreted as usual, i.e., a subclass *inherits* the terminal symbols and contained children of its supertype. Relations originating from the *relational RAG* extension are also shown.

The structure of the ROS message model shown in Figure 5.3a is obtained from the ROS system using the respective version, depending on the context in which the tool is run. The model contains (below a top-level `RosModel` node) one message (`RosMsg`), which itself contains fields, which might be constants or regular fields and use a type. This `TypeUse` specifies whether the field is an array and contains a reference to a type, which itself might be a `BuiltInType` or a `RosMsg` – the latter describes message nesting. An instance of the ROS message model for the `sensor_msgs/image` message is shown in Figure 5.4.

In fact, there are two meta-models, one for each ROS version (cf. Section 5.3.2) – the ROS2 metamodel extends the ROS1 metamodel with additional types. The aspect-oriented specification of attribute grammars [36] is used to extend the grammar. Additional elements are simply added using a ROS2 grammar module; additional attributes and attribute equations are added using a grammar aspect.

While the ROS2 communication system got a complete overhaul (internally, its messages are defined in the Interface Definition Language [122]), the *(concrete)* syntax is mostly backward compatible so that a common parser can be used. Likewise, the model (or *abstract* syntax) for

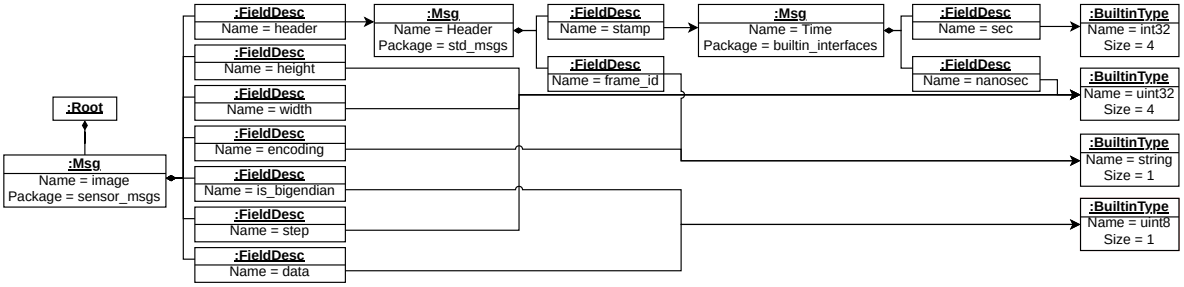


Figure 5.4: ROS message model for `sensor_msgs/Image`

ROS1 just needs minor extension to also support the features of ROS2 as shown in Figure 5.3a. The three changes (default values, bounded strings, and arrays) are highlighted. The complete support for both versions of ROS addresses Challenge 5.1: Interface Compliance.

Model-to-model transformations are used to obtain the *intermediate message model*, a generic representation of the message interface. This model is designed for efficient HDL code generation: irrelevant and redundant information contained in the ROS models are removed, and the concepts of this model are aligned with the requirements of VHDL and the AXIS format. Specifically, it contains - below a root element `VhdlModel` one `Message`, which itself contains a list of fields, which may contain atomic data types or (variable-length) streams of messages or data types. All other structural features are mapped to these features; fixed-length arrays, for example, are simply unrolled to a sequence of fields. Thus, *this model does not depend on details of the ROS message format*, creating an extension point for other message specifications, as proposed in [123]. This contributes to the solution for Challenge 5.1: Adaptivity. An instance of this intermediate model for the `sensor_msgs/image` message is shown in Figure 5.5.

The direct purpose of the *intermediate message model* is *not* to directly generate HDL code from but rather to provide input to a template engine that performs the generation. The decision to rely on a template engine also has several advantages over the direct assembly of the resulting code. Templates allow a clean separation of syntactic and semantic issues, which is especially true since, as mentioned previously, the employed engine *mustache* [124] is *logic-less* (templates do not contain computations). Thus, all computations are done before or during the construction of the template configuration files within FIRM, using its declarative static analysis capabilities. This results in not only very concise and thus easily maintainable templates but also maintainable and reviewable template configurations. Another advantage is that templates can easily be exchanged for different hardware platforms, addressing Challenge 5.1: Adaptivity. *Mustache* is used as the template engine since it is a mature tool fitting the requirements [125].

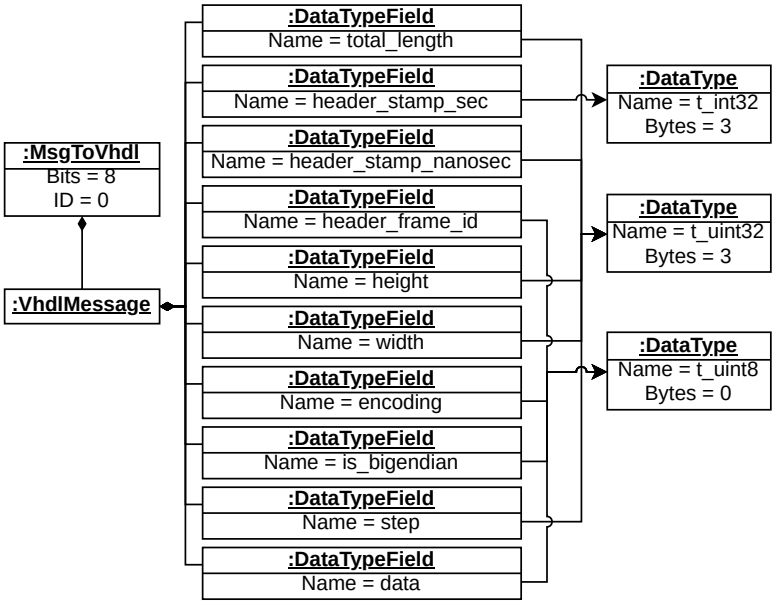


Figure 5.5: Intermediate message model for `sensor_msgs/Image`

So far, the structure of the models has been presented. Next, it is demonstrated how *attributes* are used to perform the model transformation and the analysis required for it.

5.2.4 Attributes

The computation of the attributes *bitwidth* and *StartIndex* (as shown in Figure 3.3) will serve as examples for the analysis. Listing 5.2 shows the attribute *bitwidth*, computing the bitwidth of the individual fields in a message, required during code generation. For example, Line 5 shows the equation to obtain the number of bits for a built-in data-type. In this case, `getSize` returns the number of bytes (i.e., 8, 16, 32, 64) and that is multiplied by eight to obtain the *bitwidth* (total number of bits). It is a *synthesized* attribute [116], using information from the subtree of the nonterminal it is defined for. Since the nonterminal `Field` is abstract, there are defining equations for all three non-abstract subtypes rather than for the abstract supertype itself.

A more complex attribute mentioned in Chapter 3 is *StartIndex* shown in Listing 5.3. It specifies the position of the first byte of each variable in the AXIS frame. While computing the attribute value for simple messages can easily be done by summing up the preceding variables' data type sizes, it gets more complicated in the presence of streams of sub-messages. For sub-AXIS, the initial field (containing the size) of the stream is copied from the parent stream. Their index starts again at `0` for the following fields in the sub-stream. This is computed by an *inherited attribute* obtaining information from its *context*, i.e., its parents in the abstract syntax tree [116]. In this case, the required context is the message a field is contained in and its position within the message. Thus, the attribute is *declared* for the nonterminal `Field`, but is *defined* for a `Field` which is the child of a `Message` at the position `pos`. The attribute equation uses other attributes, such as the previously presented *bitwidth*.

5.2.5 Attribute-Controlled Model Transformation

The attributes shown so far can be used to compute semantic properties of the message required in the generated VHDL code. In addition to this, attributes can also be used to perform the model transformation itself using *higher-order attributes* [120] that compute entire (sub-)trees. In the *JastAdd* tool, these attributes are also called *Non-Terminal Attributes (NTAs)*. Figure 5.6 shows the sequence of two NTAs, computing the intermediate model from the initial message model and a template configuration model from the intermediate model. These two attributes *constructVhdlModel* and *constructIpToAxisFrame* construct a subtree

```
1 // declaration of attribute bitwidth returning an int
2 syn int Field.bitwidth();
3
4 // attribute equations
5 eq DataTypeField.bitwidth() = getType().getSize() * 8;
6 eq DataTypeStreamField.bitwidth() = // ...
7 eq MessageStreamField.bitwidth()  = // ...
```

Listing 5.2: Declaration and equations for the synthesized attribute *bitwidth* for the nonterminal `Field`

```
1  // startIndex computes an 'int' for each 'Field'
2  inh int Field.startIndex();
3
4  // defining equation on the context (a 'Field' that
5  // is a child of a 'Message' at position 'pos')
6  eq Message.getField(int pos).startIndex() {
7    if (pos == 0) {
8      if (isSubmessage()) // ← another attribute
9        // for the first field in a submessage,
10       // use the startIndex of the parent
11       return containingField().startIndex();
12     else
13       // for the very first field start with 0
14       return 0;
15   } else if (pos == 1 && isSubmessage()) {
16     // for the second field in a submessage,
17     // reset the index to 0
18     return 0;
19   } else {
20       // otherwise, the startIndex is the one of the
21       // preceding field plus its size (in bytes)
22       return getField(pos-1).startIndex()
23            + getField(pos-1).bitwidth() / 8;
24   }
25 }
```

Listing 5.3: Declaration and definition of the inherited attribute *startIndex*

using the information and available attributes from the non-terminal they are defined on. Finally, a (synthesized) attribute *print* is used to obtain a string representation of the template configuration, which the template engine can use. Listing 5.4 shows (parts of) the result of this code generation for the *image* message (Figure 3.3), including values computed using the presented attributes.

5.2.6 Template-Based Code Generation

The configuration shown in Listing 5.4 (a Yet Another Markup Language (YAML) file) configures the *mustache* template engine, which finally expands a set of templates called for this work *Hardware Description Template (HDT)*. The HDTs are modified VHDL modules, adapted for template expansion following the conventions dictated by the chosen template engine (cf. Listing 5.5 and Listing 5.6).

The combination of RAG-based model analysis techniques with a template-based code generation helps to create efficient, specialized code that remains *highly portable*.

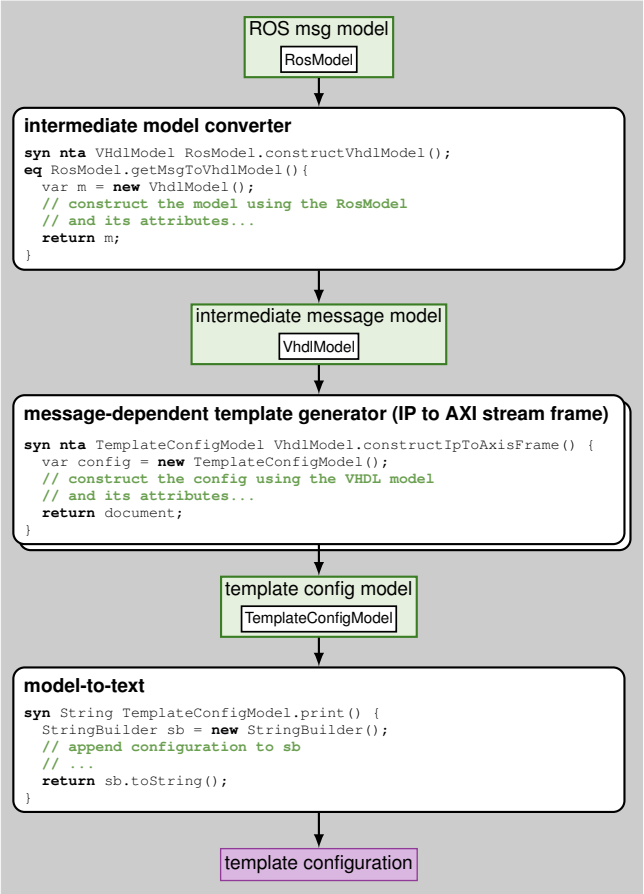


Figure 5.6: Model transformation and code generation attributes

```yaml
IP:
 name: sensor_msgs_Image_to_AXIS
 type: publisher
msg:
 BYTES: 44
message:
 isSubmessage: false
 fields:
  # ... other fields
  - simple:
    name: height
    bitwidth: 32       # see Listing 4.2
    datatype: true
    type: t_uint32
    startIndex: 21     # see Listing 4.3
    index:
     - {N: 21, MSB: 7, LSB: 0}   # N is computed with
     - {N: 22, MSB: 15, LSB: 8}  # the startIndex
     - {N: 23, MSB: 23, LSB: 16} # attribute and
     - {N: 24, MSB: 31, LSB: 24} # a byte counter
    multi: false
  # ... other fields
```

Listing 5.4: Template configuration file

5.3 Evaluation

The evaluation of the workflow presented in previous sections has been split into three parts. Section 5.3.1 analyzes the complexity of message specifications supported by FIRM. Section 5.3.2 shows the evaluation performed based on real message specifications included in both ROS versions supported and highlights the possibility of the approach also to performing logic validation of the autogenerated components. Lastly, Section 5.3.3 shows two use cases with different accelerators, as publishers and subscribers, being part of a ROS system.

5.3.1 Complexity of Specifications

The first step to ensure full support with external components (ROS-based in this case) is to analyze the characteristics of the middlewares to be interfaced. Based on the characteristics analyzed in [114], a study of the datatypes supported by ROS1 and ROS2 is performed and then compared to the features supported by FIRM, presented in previous sections. The main characteristics concerning datatypes are shown in Table 5.1. It depicts that the approach presented in this dissertation supports all characteristics of ROS1 and ROS2, and it is extendable if needed for further characteristics. Even though some of them (9, 18, and 19) are not supported by ROS, they could be easily added to FIRM if needed, following the same approach as the *extension* for a new middleware, shown next in Section 5.3.2.

It was chosen not to transmit *Null*, *Constants* and *Enums* (5, 6, and 7) to the hardware accelerators as part of the AXIS frame as those values can be directly stored as LUTs because they do not change over time. This simplifies the resulting implementations avoiding the extra logic needed to extract the values from the AXIS frame, which would imply an unnecessary increment of resource utilization. However, they are identified by FIRM, so they could be part of the AXIS frame if needed. In that case, they would be similar to characteristics 1 to 4 in Table 5.1, as they could become a new datatype considering that only their bitwidth needs to be specified.

Even though *Unions* are not supported by ROS1 nor ROS2, they could be potentially added to the proposed workflow by transmitting its width along with the data. Then it is figured out in the accelerator what it represents. It is a similar case as for *Maps*, where the key (primitive type) and the data (which could be an AXIS by itself) are transmitted, similarly to a nested message.

Once there is an understanding of the supported datatypes in both ROS versions, an analysis of their combination to form message specifications follows. The elements constituting ROS messages can be simple fields, fixed or variable-length arrays, and simple or sub-message arrays. Considering them, four levels of complexity have been identified, as shown in Figure 5.7. All their possible combinations are what lead to quite complex structures.

The simplest messages are those that contain one or multiple single elements of any of the built-in types (e.g., uint8, int16, float32). Examples of these are `height`, `width`, `is_bigendian` and `step` in Listing 3.1. Then follow the messages in level `ExpandingMessages`. These can be built-in types declared as fixed sizes arrays, which are unfolded; inlined elements of `SimpleMessage` or a combination of these two as bounded-sized arrays of messages which are unfolded and inlined. An example of the inlining is `header` in Listing 3.1, which is an

instantiation of the off-the-shelf `std_msgs/Header` message. The third level referred to `Arrays` groups variable length arrays for built-in types, such as the case of `data`, which is converted into an AXIS, as shown in Figure 3.3. Note that strings also form part of this level because their length is not explicitly defined, so they are treated as variable-length arrays (only their upper bound might be defined), as is the case of `encoding` in Listing 3.1. The upper bound of built-in type arrays is introduced in ROS2, which is also supported by FIRM. The most complex level is `VariableSubmessages`. It includes variable-length arrays of messages which are on their own, a combination of the first three levels, or the particular case of an array of strings (array of an array). In this case, a sub-instance of AXIS is generated for each element that belongs to this level of complexity.

Therefore, all these levels of complexity and their combinations need to be covered by FIRM to provide generic support for generating all sorts of complex message specifications with the proposed workflow. Specific design rules, shown in Table 5.1 and explained previously, were followed based on this to cope with the complexity and support all the combinations of messages shown in Figure 5.7 by FIRM.

Table 5.1: Supported datatypes, ✓ mark supported, ✗ unsupported, and ✢ potentially addable features.

#	Feature	*ROS1*	*ROS2*	*FIRM*	
1	Signed Integers (8, 16, 32, 64 bits)	✓	✓	✓	
2	Unsigned Integers (8, 16, 32, 64 bits)	✓	✓	✓	
3	Float/Double (32, 64 bits)	✓	✓	✓	
4	ByteBlob Type	✓	✓	✓	(using #2 and #8)
5	Null Type	✗	✗	✓	(not transmitted)
6	Constants	✓	✓	✓	(not transmitted)
7	Enums	✓	✓	✓	(not transmitted)
8	Variable Length Arrays	✓	✓	✓	
9	Multidimensional Arrays	✗	✗	✢	
10	Fixed Length Arrays	✓	✓	✓	(rolled out)
11	Maps	✗	✗	✢	
12	Optional Fields	✗	✗	✢	(new user-defined msg spec.)
13	Default Values	✗	✓	✓	(value always transmitted)
14	Unions	✗	✗	✢	
15	Message Nesting*	✓	✓	✓	
16	Data Type Inheritance	✗	✗	✗	
17	Namespaces	✓	✓	✓	
18	Typedefs	✗	✗	✢	
19	Any Type	✗	✗	✢	

*In this work, nested messages are fields with another message type as data type rather than an in-place definition of a message within another message as in [114].

5.3.2 Full ROS Support

Individual experiments were performed on all messages in the base installations from the latest three ROS1 distributions (`Kinetic`, `Melodic`, and `Noetic`) and ROS2's LTS[2] `Humble`, to demonstrate FIRM's support for the characteristics listed in Table 5.1. Three groups, namely *amount of elements*, *amount of different data types*, and *depth of nested messages*, were evaluated to showcase the diversity of these messages. These three characteristics encompass all levels of complexity shown in Figure 5.7. The results of the experiments demonstrated that the generic model-based approach employed by FIRM enables it to effectively handle `arbitrary sizes` of ROS messages, including those with `multiple data types` and `nested structures`. Figure 5.8, Figure 5.9, and Figure 5.10 show the histograms for these three characteristics based on the evaluated ROS messages used in the experiments. Note that ROS1 and ROS2 provide a `base` set of packages, including on average 150 message specifications in each distribution. Similar results were obtained from the experiments performed with all mentioned ROS versions. Additionally, different extra messages from the ROS open source community were used to evaluate the robustness of the proposed approach. All the installable packages of Noetic add up to 3102 distinct message specifications and 1346 for ROS2 Humble, as shown in Figure 5.11.

Each experiment consisted of an autogenerated project for each ROS message to run the post-synthesis simulation to validate the logic of the interfaces. These experiments were performed on all distributions mentioned before, but only the extended versions of Noetic to evaluate ROS1 messages and Humble for ROS2 are shown, as these are the two latest versions of each ROS distribution. Therein, each project included:

1. the hardware interfaces (both directions) for a specific ROS message, considered the Device Under Tests (DUTs).

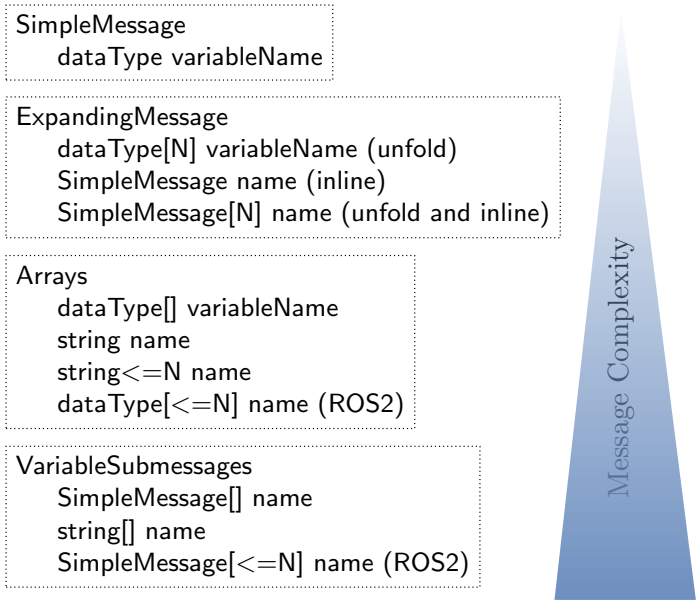
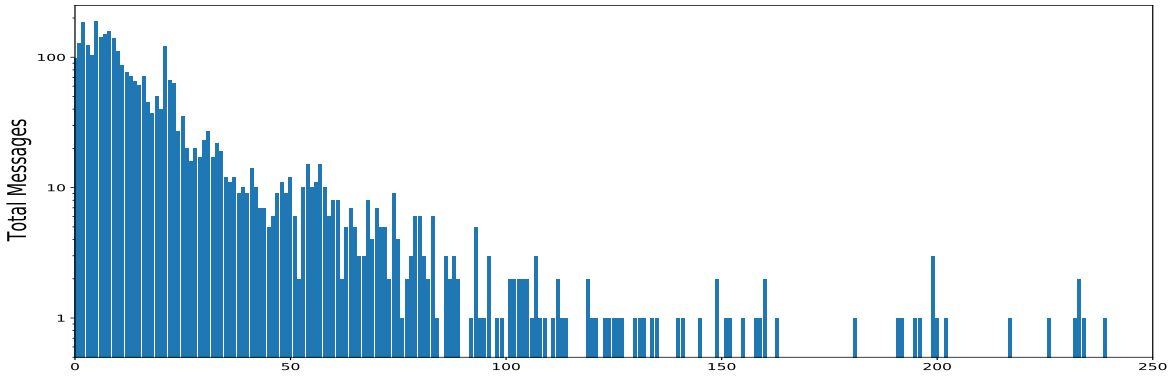
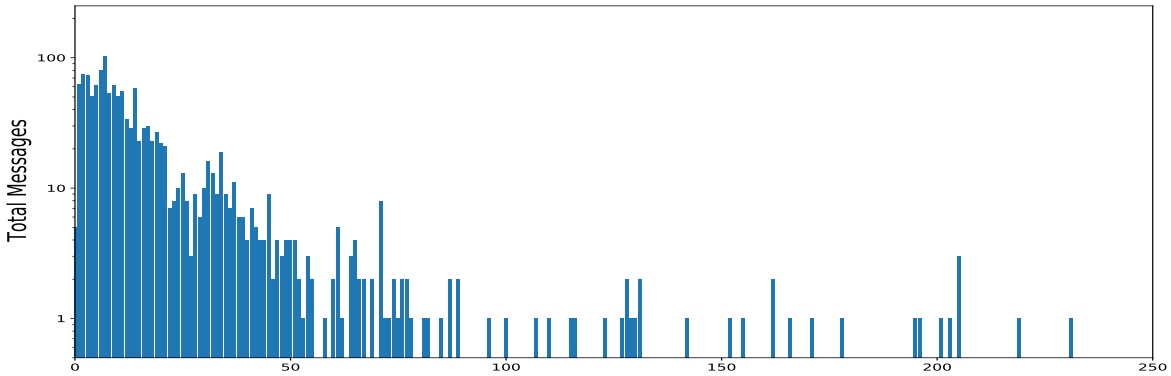


Figure 5.7: Complexity of ROS messages

[2] Long Term Support



(a) Number of contained fields in ROS Noetic (28 messages with more than 250 and up to 1234 contained fields not shown)



(b) Number of contained fields in ROS2 Humble (10 messages with more than 250 and up to 648 contained fields not shown)

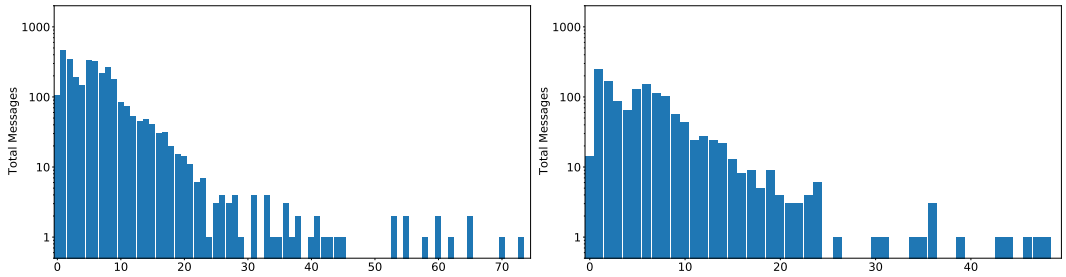
Figure 5.8: Histograms of contained fields in ROS Noetic and ROS2 Humble messages

2. the remaining components of the base architecture shown in Figure 3.1.
3. an input stimulus for the simulation.

In this case, one hardware interface acts as a subscriber and the other as a publisher. This means that the first one receives an AXIS frame (input stimulus), which depends on the ROS message being tested. This is converted into individual signals for the latter to read and generate an out AXIS frame. A successful test implies that both AXIS streams are equal. In an actual application, the accelerators that perform computation would be in between these two hardware interfaces in a publisher/subscriber combination or as shown in Figure 3.1.

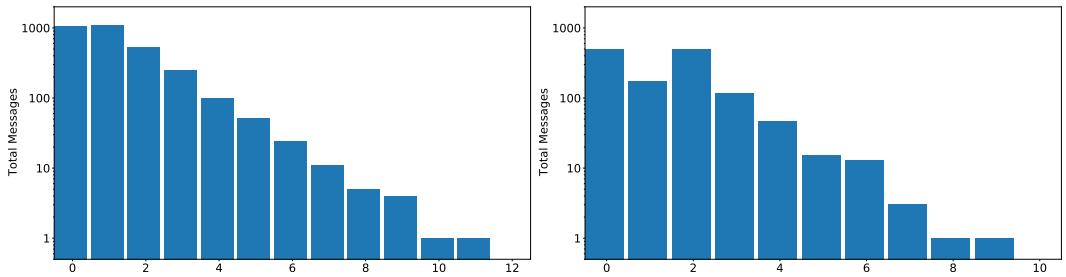
Both DUTs are generated from the combination of template configurations obtained from the intermediate representation in FIRM and the HDTs, as depicted in Figure 5.2. Each project for every ROS message includes the same `accelerator-related` components (Chapter 3), to emulate an actual application scenario, as demonstrated in the use cases in Section 5.3.3, rather than just using the DUTs alone.

The generation of the input stimulus is also done with a combination of a new template configuration obtained from *message models* in FIRM and a new set of templates (cf., Figure 5.2, *integration test extension*), tailored to ROS code in C++. Hence, a generated native ROS node populates the fields of the corresponding message being validated and stores it as a series of bytes for the VHDL testbench to use as the input stimulus. Note that it is advantageous to



(a) Distinct types in ROS Noetic (b) Distinct types in ROS2 Humble

Figure 5.9: Histograms of distinct data types in ROS Noetic and ROS2 Humble messages



(a) Nesting depth in ROS Noetic (b) Nesting depth in ROS2 Humble

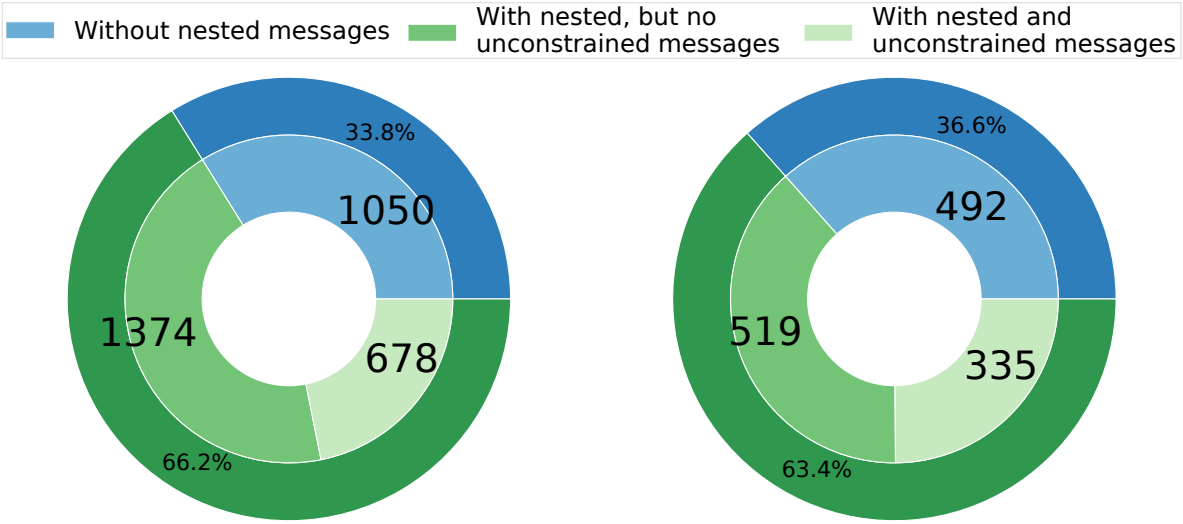
Figure 5.10: Histograms of nesting depth in ROS Noetic and ROS2 Humble messages

have already available the intermediate representation of the ROS message to be evaluated. However, as specific fields in the message may have variable lengths (arrays or strings) or may be nested messages, this needs to be accounted for in order to generate a meaningful input stimulus. For these cases, besides considering the data type (e.g., uint8, uint16, string, float64), a random length for an element that requires it is generated. Consequently, `arbitrary message lengths` are evaluated:

1. for cases in which messages do not include fields with variable length, the variety of different off-the-shelf messages (Figures 5.8 to 5.10) ensures multiple message lengths.
2. for cases in which unconstrained elements (Figure 5.11) or nested messages are included (Figure 5.10).

These two points ensure the extensive evaluation of arbitrarily large messages, whether they are from ROS1 or ROS2. Figure 5.8, Figure 5.9 and Figure 5.10 show a comprehensive coverage of tested messages composed by `multiple distinct variable types`, ensuring the capabilities to also support ROS messages with multiple data types.

It can be seen in the histograms that there are more messages with up to 80 fields for ROS Noetic and 50 for ROS2 Humble (Figure 5.8), including around mainly 20 distinct types for both of them (Figure 5.9). Figure 5.10 shows that multi-level nesting is used in both ROS versions with several levels of depth (up to 11 for Noetic and 9 for Humble), which is supported as well by FIRM. The differences in the tuples of Figure 5.8, Figure 5.9 and Figure 5.10 are because there are currently more projects with ROS, but the transition to ROS2 is ongoing. These results show that the generic approach proposed in this dissertation is appropriate to support large or complex messages with many distinct fields or multiple levels of nested messages, regardless of the ROS version. However, FIRM is not limited only to ROS1 or ROS2,



(a) ROS Noetic (b) ROS2 Humble

Figure 5.11: Amount of ROS and ROS2 messages with and without nested message

as the same process to migrate the templates from the first to the second version can be followed for other types of message specifications.

As an early evaluation phase, a systematic approach was followed during the design process to *manually* generate a set of message specifications to cover all of the possibly infinite numbers of them. This left out cases that were not considered, which is why the presented evaluation solution relies on all installable ROS packages, which allows to automate the process of validation of real message specifications deployed in multiple applications. An open point to explore is how to generate representative datasets for these experiments to validate the logic design of the generated hardware component based on the information provided by Table 5.1. The stimulus for each evaluated component is obtained with a stimulus generator (a native ROS node) for each message that automatically creates a stimulus with all constraints met. Currently, only one random message is generated as a stimulus, but this can be extended to generate more stimulus for each message, randomly or systematically, to extend coverage. This implies the generation of multiple stimulus for each message, which would increase the total time to run all the experiments.

The first iteration [9] took over 41 hours with one stimulus for every 2295 for ROS Noetic messages and ROS2 Foxy[3] with 150 messages. The toolchain and the evaluation process were improved and optimized, and even though more messages were added to ROS Noetic (3102 in total) and 1346 for ROS2 Humble, the entire evaluation took 13.8 and 3.6 hours respectively.

Extension for ROS2 support

The evaluation was performed on both ROS versions (Noetic and Humble), possibly due to the simplicity of extending support for ROS2 from the existing solution for ROS1. Due to the model-based design of FIRM, which derives the intermediate message model, and the

[3]Previous ROS2 LTS version

ability to design tailored templates, support for ROS2 can be easily achieved. Using *.msg* for message specifications brings backward compatibility. Nevertheless, the communication scheme in ROS2 is the main characteristic of this new version[4] and the serialization of ROS messages differs slightly. This leads to a new set of HDTs adapted to take it into account for ROS2 support.

Listing 5.5 and Listing 5.6 show a snippet of the HDTs for ROS1 vs. ROS2 respectively. It is possible to see there the specific syntax of the used template engine `mustache`. `Tags` (as called in mustache) are expressed between curly brackets (i.e., line 1 in Listing 5.5). These are the fields that FIRM generates in the template configuration (Listing 5.4). Comments can be written by prefixing an exclamation sign to the tag (i.e., line 1 in Listing 5.5), and they will not be generated in the resulting artifacts. Mustache checks whether the tag is present in the template configuration when the symbols # or ˆ (i.e., line 3 or 6 in Listing 5.6) are present. They are used conditionally to generate the contained code in a block in the resulting artifact. The value assigned to a field will be generated when the tag is only expressed in the template, such as `N` in line 2 of Listing 5.6.

ROS2 includes extra logic for padding, necessary to meet any desired memory-alignment requirements. It is relative to the *first byte* of a variable, and it is determined by computing the modulo between the total number of currently streamed bytes (s_inputs in this example) and the bitwidth of each element (size). The differences between Listing 5.5 and Listing 5.6 and how they can be easily expanded highlight the benefits of our approach and how Challenge 5.1: Adaptivity was tackled. Minor changes to include the field *padding* in the intermediate message model and a complementary set of HDTs (extended from the ones for ROS1) were only needed to provide support also for ROS2.

Table 5.2 shows the difference of *Lines of Code (LoC)* for both sets of HDTs. Note that each set of templates is composed of multiple files (called partials) to modularize their design, improving the maintainability.

As expected from the snippets shown in Listing 5.5 and Listing 5.6, the HDTs for ROS2 have more lines of code than the ones for ROS1. Not only the `message-dependent` hardware components for publishers (`message` specification to AXIS) and subscribers (AXIS to `message` specification) are modified. The template to generate the stimulus is also updated, mainly due to new built-in types included in ROS2.

5.3.3 Use Cases

Two use cases were developed to prove the feasibility of the workflow presented in previous sections. The first one is related to image processing, and the second one is an FPGA-

```
1 {{^isSubmessage}}   {{!mustache element}}
2 if(s_counter={{N}}) then
3     s_counter <= s_counter + '1';
4 end if;
5 {{/isSubmessage}}
```

Listing 5.5: Snippet of a HDT for ROS1

[4] `http://design.ros2.org/articles/changes.html`

```
1  {{^isSubmessage}}
2  if(s_counter={{N}}) then
3      {{^padding}}    {{!same as for ROS1}}
4      s_counter <= s_counter + '1';
5      {{/padding}}
6      {{#padding}}    {{!extension for ROS2}}
7      if((to_integer(unsigned(s_inputs))
8          mod {{size}}) = 0) then
9          s_counter <= s_counter + '1';
10     end if;
11     {{/padding}}
12 end if;
13 {{/isSubmessage}}
```

Listing 5.6: Snippet of a HDT for ROS2

Table 5.2: Lines of Code of ROS1 and ROS2 HDT.

HDTs	Files	Total Lines of Code	
		ROS1	ROS2
AXIS to msg	10	292	369
msg to AXIS	9	296	383
Stimulus	2	177	236

based mobile robot. Both of them are part of a ROS architecture with different message specifications.

Image Processing

The setup for this use case consists of a publisher/subscriber set on the FPGA as well as on a PC. The sequence interaction of them is shown in Figure 5.12. The latter one publishes a webcam feed (640x480@30fps) on a topic that the FPGA subscribes to. Then, it processes the raw image to publish on a different topic for the PC to subscribe to. The hardware accelerator (Sobel filter) is based on the open source "High-Level Synthesis FPGA Library for Image Processing" (*HiFlipVX*) [14], which offers a large set of different functions that can be combined in a "building blocks" fashion. The ROS `message` specification chosen is `sensor_msgs/Image` (Listing 3.1), a complex one because it includes different data types, constrained and unconstrained sized variables. The process begins by writing the *Configuration File* (Listing 5.1), which is the input of the workflow. As previously mentioned, specific details of the platform (lines 2 to 5) are needed. Then, follows the information related to the accelerators to be interfaced. FIRM takes it as its input, as well as the ROS `message` specifications listed there to generate the required interfaces (`message-dependent` components). Additionally, `TCL` scripts for Vivado to build the whole project including the components shown in Figure 3.1 are generated. The details of these additional artifacts generated are covered in Chapter 6.

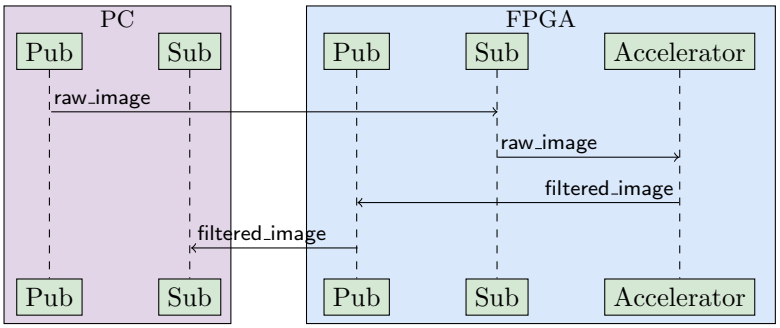


Figure 5.12: Image processing use case sequence

FPGA-Based Mobile Robotic Platform

A skid-steer mobile robot with four DC motors and quadrature encoders is used and controlled with a combination of VHDL and HLS IP cores. They receive velocity commands or send the robot's current state through ROS messages. The direction of rotation and speed of each wheel is obtained via VHDL IP cores as well as PWM signals to set their speeds. A PID (implemented in HLS) controls the speed of the wheels. The ROS `message` specification used here is `geometry_msgs/Twist` as it is composed by linear and angular speeds in three axes (x,y and z). The presented workflow allows to seamlessly generate a new hardware architecture by only modifying the configuration file (Listing 5.1), specifying the platform (Lines 2 to 6) and the name of the ROS `message` specification in Line 9.

Experiments

Tackling the challenge to have an *integrated workflow* allows performing experiments on two FPGA-based SoC (Zynq UltraScale+ and Zynq 7000) with ease. Note that the family is derived from the part specification, as shown in Line 5 in Listing 5.1. Zynq-7000 or Ultrascale are the supported ones and are extendable if needed.

Both experiments rely on FreeRTOS (which provides an off-the-shelf TCP/IP stack) running on one ARM core. A DMA is used to exchange data between PS-PL, imposing a bandwidth limitation. Its clock frequency is set to 300MHz, and considering the design decision of using 8 bits for data transfers; the maximum achievable throughput is $2.4 Gb/s$. Despite of this, experiments showed that almost $50 fps$ for $1920x1080$ image resolution can be achieved if needed, as shown in Table 5.3. Therefore, the `accelerator-related` and `message-dependent` components do not bring significant overhead in terms of execution time for the hardware accelerators. The execution time is slightly increased with the generated components due to the latency of a few clock cycles of some of them, for example, the schedulers to make a decision as to which accelerator to give the grant to. However, this is not because of the autogeneration of the components but the design of some of them on their own, that they introduce a few clock cycles latency. The latency would be increased regardless they are autogenerated or coded manually, as their design is done manually beforehand, and the autogeneration focuses more on reproducibility and scalability.

If needed, it is possible to achieve a larger bandwidth by increasing *tdata*'s width (up to 256 bits). However, a trade-off between the extra logic needed and the resource utilization would

Table 5.3: Execution time with and without generated components.

Resolution	Execution time for the accelerator [ms]	Execution time with generated components [ms]	Frames per second
640 x 480	3,07	3,08 (+0,25 %)	325
800 x 600	4,80	4,81 (+0,20 %)	208
960 x 720	6,91	6,94 (+0,40 %)	144
1024 x 768	7,86	7,88 (+0,19 %)	127
1280 x 720	9,21	9,23 (+0,15 %)	108
1920 x 1080	20,73	20,76 (+0,11 %)	48

have to be evaluated. The overhead introduced by the autogenerated components in terms of resource utilization and performance is evaluated. Table 5.3 shows the execution time of the sobel filter by itself and embedded and as part of the process shown in Figure 5.12. It can be seen the negligible increment in the execution time the `accelerator-related` and `message-dependent` components add.

Resource Utilization

Table 5.4 shows the difference of resource utilization of both use cases. The *PS-PL Interconnection* will remain unchanged regardless of the application when a software solution is used for the communication outside the FPGA. *Autogenerated Components* refer to the `Manager` plus the hardware components based on the message specifications (`message-dependent`) used for each use case. It can be deduced that they will not introduce significant overhead to a design in terms of resources. The differences between use cases are because:

1. the `message` specification for image processing has three unconstrained variables (converted into AXIS) compared to only constrained ones for the mobile robot. This increases the logic (hence LUTs) for the former one and latches (FFs) for the latter one.
2. There is only one HLS IP core combining the splitting of the RGB channels and computing a Sobel filter on each of them, so it is optimized. The mobile robot includes multiple VHDL and HLS IP cores, mostly performing arithmetic operations, which corresponds to DSPs usage.

Automation

Listing 5.1 shows what the configuration file for the image processing use case looks like. It takes the targeted platform (FPGA and board), as well as the ROS message specification[5]. It can be seen that only a few lines are needed to model the system compared to multiple extensive VHDL modules (Table 5.5). This also reduces the probability of errors and increases the consistency of new designs. The time needed to generate a hardware architecture is

[5]It can include multiple message specifications, publishers, and subscribers.

Table 5.4: Resource utilization for both use cases

Image Processing (Zynq UltraScale+)	LUT	FF	BRAM	DSP
PS-PL Interconnection	6735	7374	2	0
Autogenerated Components	759	321	0	0
Hardware Accelerators	792	973	3	0
Complete Implementation	8286	8668	5	0
FPGA-based Mobile Robot (Zynq 7000)	**LUT**	**FF**	**BRAM**	**DSP**
PS-PL Interconnection	6735	7374	2	0
Autogenerated Components	456	126	0	0
Hardware Accelerators	29671	21539	0	578
Complete Implementation	36862	29039	2	578

reduced to a matter of minutes. As stated previously, only minor configuration file adaptations are needed to generate new hardware components for different applications automatically.

The benefits of the proposed approach are that even though there is a similar effort (in terms of lines of code, shown in Table 5.5) between writing VHDL and HDT, the latter one has to be written *only once* and can be reused for *multiple* use cases. VHDL, on the contrary, has to be manually adapted every time for different ones. A designer would need to invest time to adapt each of them individually. This could quickly become an issue when something is unintentionally neglected, or errors are introduced by inconsistently changing existing parts of the VHDL model.

Lastly, the approach facilitates the generation of test benches tailored for any ROS message (shown in Section 5.3.2) or any other type of specification, tackling Challenge 5.1: Interface Compliance. This simplifies a complete hardware architecture validation process with a single specification as the only input parameter.

Table 5.5: Lines of code written once for all use cases, and additional written/generated code for each individual use case

File type	Written Once	Use cases	
		Image Processing	Mobile Robot
Mustache TCL	30	0	0
Static TCL	52	0	0
Mustache VHDL (HDT)	588	0	0
Static VHDL	171	0	0
Config. File	n/a	13	13
Generated TCL	n/a	64	64
Generated VHDL	n/a	432	353

5.4 Summary

This chapter presented a model-based approach to automatically generate hardware components acting as interfaces for FPGAs accelerators handling the compute-intensive tasks for robotic applications. A simple specification of the expected system is introduced, which is the only input needed to generate the complete hardware architecture. Hardware components acting as interfaces for the accelerators are obtained from message specifications, supporting arbitrary ROS messages. As demonstrated, the middleware support can be extended, if required, with minimal effort.

An intermediate message representation is obtained from the input specification, detailing the type of interface required. In addition to a set of templates derived from a HDL module, this is used to generate the corresponding hardware components to interface the accelerators automatically.

All message specifications included in both ROS versions (all latest distributions) were evaluated. Besides, two use cases show the advantages of our approach by integrating an HLS image processing IP core as well as a combination of customized HLS and VHDL modules for an FPGA-based mobile platform into a ROS architecture. The first one required only 13 lines of code for the input specification of the workflow to deploy the entire system. It only took minor changes to some lines (rather than entire VHDL modules) to generate the second use case, even on a different FPGA family.

Chapter 6 describes the extension of FIRM and the details involved in data-type and data-flow analysis for the generation and automatic deployment of the entire architecture, considering the `accelerator-related` and `message-dependent` components.

6 Model-based Generation of Hardware/Software Architectures for Robotics Systems

Robotic systems compute data from multiple sensors to perform several actions (e.g., path planning, object detection). FPGA-based architectures for such systems may consist of several accelerators to process compute-intensive algorithms. Designing and implementing such complex systems tends to be an arduous task. This chapter extends the workflow presented in Chapter 5, focusing on the modeling to generate architectures for such applications, compliant with existing robotics middlewares (e.g., ROS, ROS2). The challenge is to have a compact yet expressive system description with just enough information to generate all required components and integrate existing algorithms. This system model must be generalizable, so it is not application-dependent and must exploit the benefits of FPGAs over software solutions. Previous work mainly focused on individual accelerators rather than all components involved in a system and their interactions. The proposed approach exploits the advantages of MDE and model-based code generation to produce *all* components, i.e., message converters (`message-dependent components`) acting as middleware interfaces and wrappers to integrate algorithms. Data type and data flow analysis are performed to derive the necessary information to generate the components and their connections.

6.1 Challenges and Goals

The range of robotic applications has been increasing lately, from manufacturing [2], collaborative robots interacting with humans [3], biomedicine [4], drones [5] as well as mobile robots [6], to name a few. Due to the wide range of applications, robotic platforms are becoming more complex as more heterogeneous data from different types of sensors needs to be processed, preferably concurrently, to meet real-time constraints. An architecture should facilitate the development of robotic systems by providing helpful constraints on the design and implementation of the desired application without being overly restrictive [126]. However, designing FPGA-based architectures for such systems tends to be an arduous process as it requires low-level hardware knowledge and a long and complex design process. Even though the proven advantages of FPGAs for robotic applications [127, 128, 129], porting them from software to embedded hardware platforms or accelerating parts requires the creation of suitable interfaces. This often means the re-design of several parts of the

applications. Lastly, the interconnection of multiple components for complex applications (i.e., multiple accelerators) turns into an error-prone process.

This chapter addresses the modeling approach to generate architectures for robotic applications in FPGAs. The main *research questions* to answer are how to generate all required components for such architectures from a *holistic model* and how that model should be defined. These bring some requirements:

- Requirement 6.1: the description should be compact, concise, but expressive enough to contain the necessary information to *derive* the system's components and their relations.
- Requirement 6.2: the approach must be generic rather than application-specific.
- Requirement 6.3: it must exploit the benefits of FPGAs over software solutions.

Three main challenges arise:

- Challenge 6.1: Obtain the explicit and derive the implicit information from the *system specification*.
- Challenge 6.2: The *system specification* has to be a *compact and meaningful* description so writing it is not as cumbersome as deploying the system manually.
- Challenge 6.3: There has to be an *understanding* of the specifications of interfaces to generate the compliant components and the relations among each other.

To address these challenges, the main goals of this chapter are:

- `Model Analysis`: A comprehensive *analysis* of the *system specification* to *derive* the *holistic model* that includes all the components to generate, their interfaces and how they all interact among each other.
- `Interfaces`: Wrappers for accelerators based on middleware specifications to ease their integration.

6.2 Code Generation Workflow

A typical robotics system is composed of different types of components. They can be CPUs, accelerators, and those that act as interfaces between the first two (`message-dependent components`). The concepts shown in this work follow the Zynq[1] device model with a PS and a PL sharing data via DMA. However, this is not limited as the CPU support can be extended (e.g., soft-cores) or removed if not needed, and the toolchain proposed in this chapter takes this into consideration.

An example of a modeled and later generated FPGA-based robotics system following the concepts of this dissertation is shown in Figure 6.1. It consists of a *subscriber converter* to receive a quaternion, an *accelerator* to compute the conversion to Euler angles, and a *publisher converter* to broadcast the result. Note that even though the aim is to generate complex systems, including multiple accelerators with their middleware-based interfaces,

[1] ZYNQ is a trademark of Xilinx, Inc

only one accelerator with its corresponding `message-dependend` components are shown for simplicity. AXIS slave (S_AXIS) and master (M_AXIS) connect to DMA through the `Manager` to schedule transactions between PS and PL [9]. They exchange data as the middleware (ROS in this case) runs on the ARM processor, playing the role of the `Communication Interface` from Figure 3.1. It is mainly used to register the hardware subscriber and publisher to ROS master (Figure 2.11) and exchange data with external components. The subscriber and publisher are the `message-dependent` components, generated with FIRM, introduced in Chapter 5. It is then imperative to understand the characteristics of all components and the interaction among them to generate the corresponding artifacts to build such systems based on a given specification.

Listing 6.1 shows how to describe such a system for the proposed workflow. The interfaces of the *accelerator* (Line 15 and Line 18) include a *message* type. This is used to generate wrappers with the desired signals corresponding to that message type for the components doing the computation. In this case, the *accelerator* is an HLS component (Line 10), so the equations for the conversion are defined in a *.cpp* file (Line 11). VHDL is also supported, and additional HDLs can be added with *templates*, as explained previously. A software implementation is also possible by changing the *type* to `ROS-SW`. How to specify all components and how they interact with each other is shown from Line 21. Similar to the accelerators, the message type for *publishers* and *subscribers* must be defined. Lastly, the output of each component must be declared as *outgoing*, defining the destination block. Like so in a compact specification, the characteristics of accelerators, their interfaces, and how to establish the communication for incoming and outgoing data have been defined.

Multiple additional components are involved in such architectures besides the converters and accelerators. They are the ones that depend on the integrity of the system (i.e., Manager, DMA), depending on how many converters and accelerators are involved. These components are not part of the *system specification* as they are not generated, but their configuration is *derived* from it, as explained in Section 6.4.4. Additionally, *tailored scripts* are needed to deploy the entire architecture. The workflow of the proposed toolchain is shown in Figure 6.2, with Listing 6.1 as an example of a *system specification*.

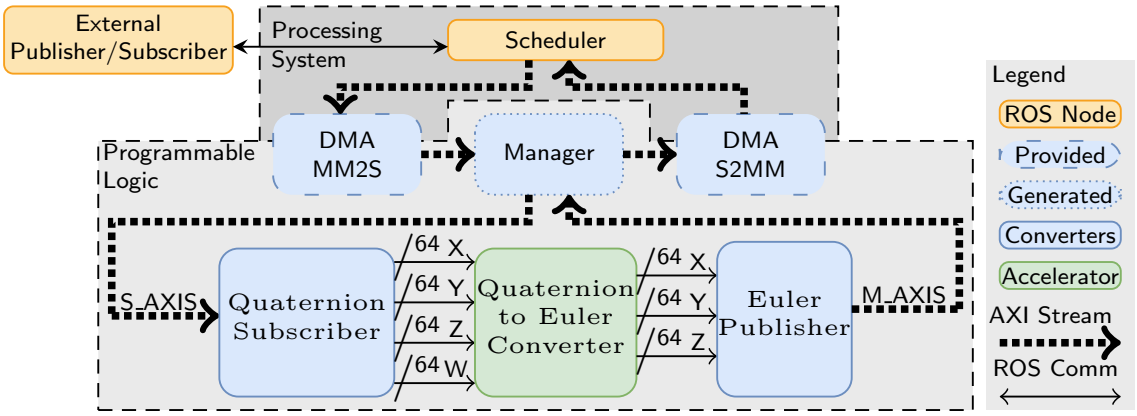


Figure 6.1: Quaternion to Euler converter with ROS interfaces

6.2.1 Model Analysis

The model-based approach proposed here helps to have a complete understanding of the desired system via model analysis. This is particularly important to generate and deploy complex systems (e.g., multiple accelerators interconnected among them with full middleware-based interfaces) as a simple process. The information to generate the different converters, wrappers for accelerators, and tailored scripts is deduced *only* from the *system specification*. All required information that is not explicitly defined (e.g., total components to manage transactions between PS and PL) is derived by doing data-type and data-flow analysis of the message types and connections of the components. All individual connections (at signal level) among all blocks, based on the specification and their interfaces, are also inferred.

```
1  project:
2    name: QuaternionToEuler
3    fpgaPartpart: xc7z020clg400-1       # Using the part can be derived
       Zynq (xcXXX)
4                                        # or UltraScale (xczcuXXX)
5  # Accelerators can be:
6    # Provided: sources (need to be exported) or already exported
7    # Generate: Wrappers from msg for HLS or VHDL (or any other template -e
       .g., verilog, systemverilog-)
8  accelerators:
9    - name: QuaternionToEuler_type
10     type: HLS                         # can be HLS, VHDL or ROS-SW
11     sources: ./QuaternionToEuler.cpp
12     interface:
13       input:
14         - middleware: ROS
15           message: geometry_msgs/Quaternion
16       output:
17         - middleware: ROS
18           message: geometry_msgs/Point
19
20 # Definition of all components and their relations
21 blocks:
22   # Subscriber
23   - name: QuaternionToEuler_sub         # converter
24     type:                               # ROS > accelerator
25       middleware: ROS
26       mode: subscriber
27       message: geometry_msgs/Quaternion
28     outgoing:                           # can have many destinations
29       - name: QuaternionToEulerConverter
30   # Accelerator
31   - name: QuaternionToEulerConverter    # accelerator of the type
32     type: QuaternionToEuler_type        # defined in line 8 and it
33     outgoing:                           # can be used multiple times
34       - name: QuaternionToEuler_pub
35   # Publisher
36   - name: QuaternionToEuler_pub         # converter
37     type:                               # accelerator > ROS
38       middleware: ROS
39       mode: publisher
40       message: geometry_msgs/Point
```

Listing 6.1: System specification for a Quaternion to Euler system

All this *derived* information is expressed in an extended and detailed version of the *system specification*, as a *template configuration* for the *template engine* to generate the desired artifacts tailored for the specified system.

6.2.2 Template Engine

The *template engine*[2] along with templates are used to generate the *intermediate artifacts*. A *template* is a generic source code that resembles the expected artifact. It is expanded with given specifications (*template configuration*) accordingly to the needs (e.g., names, bit widths). These templates are included in the toolchain. They are *coded once* and are re-used for any *system specification*. There are multiple ones involved, according to the *intermediate artifact* to generate. These can be for HLS or VHDL sources (e.g., converters) or *tailored scripts* as configurations for *vendor dependent* tools to generate the expected components. New *templates* can be added to the toolchain with ease to extend it for new components, additional hardware description methods (e.g., Verilog), or scripts for different vendors.

Listing 6.2 shows an example of a template. In this case, it is used to generate shell scripts to obtain an IP block that can be used in a Vivado's block design to instantiate the different `message-dependent` components. Note that Line 12 and Line 23 are used to expand the templates accordingly to the template configuration derived from the model analysis. `nameBD` is the IP block's name that will be used to instantiate the `message-dependent` component in any block design (hence BD) and `nameMW` is the name that the *message specification* receives for each middleware (hence MW). It can be the case where *the same* `message-dependent` component is used in the same block design. Data-flow analysis is further relied on to avoid duplicate names by appending an incremental ID to every new converter that is

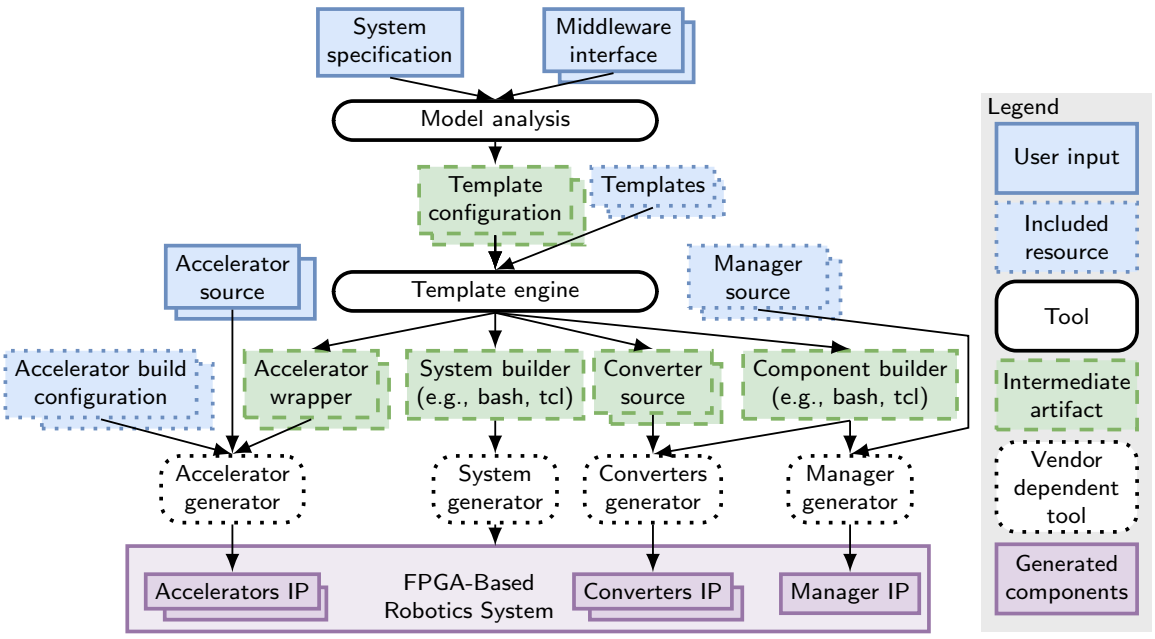


Figure 6.2: Extended toolchain workflow for the generation of HW/SW architectures

[2] Mustache—Logic-Less Templates, `https://mustache.github.io`

instantiated. This list of IDs for each `menssage-dependent` component is *derived* from the *system specification*.

6.2.3 Artifacts Generators

They take the *intermediate artifacts* to build and deploy the entire system. There are two types. Those that generate components (i.e., accelerators, converters), and the *system generator* which does not generate components but uses them. The latter one takes a set of *tailored scripts* for each application and the information of a (vendor-dependent) targeted platform. It deploys all the generated components and the derived ones (e.g., Manager, DMA). Additionally, as their interactions have been derived (each individual signal), it connects all of them accordingly, as specified in the *template configuration*. In this work, there are bash scripts that manage different tcl scripts for Vivado and Vivado HLS tools to import the generated and

```
1 #!/bin/bash
2
3 sources_converters_path=$1
4 vivado_prjts_path=$2
5 ip_repo_path=$3
6
7 ## Create directories
8 mkdir -p $sources_converters_path
9 mkdir -p $ip_repo_path
10 {{#Converters}}
11 {{#AXIS_to_msg}}
12 mkdir -p "$ip_repo_path/{{nameBD}}"
13 {{/AXIS_to_msg}}
14 {{#msg_to_AXIS}}
15 mkdir -p "$ip_repo_path/{{nameBD}}"
16 {{/msg_to_AXIS}}
17 {{/Converters}}
18
19 ## Create message-dependend components with FIRM
20 ## Subscribers
21 {{#Converters}}
22 {{#AXIS_to_msg}}
23 java -jar FIRM.jar axis2msg {{nameMW}} $sources_converters_path
24 {{/AXIS_to_msg}}
25 {{/Converters}}
26 ## Publishers
27 {{#Converters}}
28 {{#msg_to_AXIS}}
29 java -jar FIRM.jar msg2axis {{nameMW}} $sources_converters_path
30 {{/msg_to_AXIS}}
31 {{/Converters}}
32
33 ## Create new vivado project to import all sources and export the IPs
34 vivado -mode batch -source build_converters.tcl -tclargs "Converters" "
       $vivado_prjts_path/Converters" {{#project}}{{#platform}}"{{part}}"{{/
       platform}}{{/project}} $sources_converters_path $ip_repo_path
```

Listing 6.2: Mustache template to generate script that uses FIRM to generate all message-dependend components

provided sources (i.e., .cpp for the accelerators, .vhd for the converters and manager) and export them as IPs accordingly to deploy the desired *holistic system*.

Listing 6.3 shows the resulting shell script obtained with the derived template-configuration and the template shown in Listing 6.2. As there are two `message-dependent` component in the example shown in Figure 6.1, FIRM is called to generate the converters from AXIS to message and message to AXIS in Line 15 and Line 17. Then, another generated script from a different template is called in Line 20 that takes the VHDL generated with FIRM to export them as IP blocks to be instantiated in any block design.

6.3 Code Generation Challenges for HW/SW Architectures

Three main challenges arise when generating the architecture proposed in this dissertation, which are described below.

6.3.1 Concise Holistic Model

An important aspect is to have a concise but expressive description of the system (CH2), as shown in Listing 6.1. This means there has to be a mechanism to *include* or *exclude* signals from one component to another. Examples of these are shown in Listing 6.4 (Line 18 and Line 20). These keywords are analyzed to determine *which signals* corresponding to a message specification (Line 15) should be connected to which component. They can be individual

```
1  #!/bin/bash
2
3  sources_converters_path=$1
4  vivado_prjts_path=$2
5  ip_repo_path=$3
6
7  # Create directories
8  mkdir -p $sources_converters_path
9  mkdir -p $ip_repo_path
10 mkdir -p "$ip_repo_path/AXIS_to_geometry_msgs_Quaternion"
11 mkdir -p "$ip_repo_path/geometry_msgs_Point_to_AXIS"
12
13 ## Create converters with FIRM
14 # Subscribers
15 java -jar FIRM.jar axis2msg geometry_msgs/Quaternion
     $sources_converters_path
16 # Publishers
17 java -jar FIRM.jar msg2axis geometry_msgs/Point
     $sources_converters_path
18
19 ## Create new vivado project to import all sources and export the IPs
20 vivado -mode batch -source build_converters.tcl -tclargs "Converters" "
     $vivado_prjts_path/Converters" "xc7z020clg400-1"
     $sources_converters_path $ip_repo_path
21
```

Listing 6.3: Resulting shell script to generate IP blocks for message-dependend components

signals as well as sub-messages. Analyzing the structure of the message definition allows for filtering and deriving the desired signals from one component to another.

6.3.2 Dynamic Frame Length

Listing 6.4 shows the specification of a system which contains an HLS accelerator of an image processing application compliant with a *sensor_msgs/Image* message from ROS. This message includes a string (i.e., frame_id) which varies with every new frame, and the image itself could also vary depending on the application (e.g., image upscaling/downscaling). Hence, the number of bytes for the *publisher* to transmit (AXIS frame length) can change dynamically. Figure 6.3 depicts the generated components for such system. It contains the *subscriber* and *publisher converters* (to send/receive the image message from/to the PS over DMA), and the image processing application itself provided by the user (Line 4). The transmission of the message through the *publisher converter* cannot start unless the total number of bytes (frame length) to transmit is known. Hence, the *frame length* component computes this at runtime. Considering the case that the message *may not transmit* all of its fields, or the ones containing fields that *change their length dynamically* (e.g., strings), the total length cannot be known at compile time. Therefore, a tailored component to compute the frame length of each *publisher* dynamically is generated when needed and added as shown in Figure 6.3.

```
1  accelerators:
2    - name: GrayScale
3      type: HLS
4      sources: ./grayScale.cpp
5      interface:
6        output:                # Same for input
7          - middleware: ROS    # (simplified due to space)
8            message: sensor_msgs/Image
9            include: ["height", "width", "data"]
10 blocks:
11   - name: ImgFilter_sub      # converter
12     type:                    # ROS > accelerator
13       middleware: ROS
14       mode: subscriber
15       message: sensor_msgs/Image
16     outgoing:
17       - name: ImgFilter_pub
18         exclude: ["height", "width", "data"]
19       - name: GrayScale
20         include: ["height", "width", "data"]
21   - name: GrayScale_acc      # accelerator of the type
22     type: GrayScale          # defined in line 2
23     outgoing:
24       - name: ScaleDownNearest_acc
25         include: ["height", "width", "data"]
26   - name: ImgFilter_pub      # converter
27       type:                  # accelerator > ROS
28         middleware: ROS
29         mode: publisher
30         message: sensor_msgs/Image
```

Listing 6.4: Snippet of the connections between accelerator and publisher converter

Software implementations have access to large memory blocks, and the entire message is constantly available. This is not possible on the hardware side as data is *streamed*, which makes it necessary to have a mechanism to compute the total bytes in each frame as they can change dynamically. The fields of a message involved in this computation are derived by analyzing Listing 6.4. This will provide the individual lengths of dynamically changing fields needed as *inputs* for this new component to compute the *publisher*'s frame length. The fixed-sized field lengths are computed in the analysis, as these are known at compile time.

The algorithm shown in Listing 6.5 computes the length of a message from its contained fields using the helper methods `FieldLength` to compute the length of a field and `TypeLength` to compute the length of a type. Fields can be arrays of variable length not known before receiving a message. Thus, signals connected to AXIS must be used to obtain the length at runtime using the *signal()* function. Note that because arrays (and messages) can be nested, but their contents are not uniform, each information taken from a signal must be obtained at the right time during the reception of the message. This means the `TotalLength` can only be computed once the last size signal of an array within the message has been received. Because the size signals are evaluated at different times, parts of the message might need to be buffered [130], which is also inferred at the *Model Analysis* stage.

Even though the buffer size can vary from message to message or even from different frames for the same message, the WCET concept can be followed. The *transfer time* of the accelerators becomes the WCET. The metrics used for evaluating the proposed schedulers in Section 4.3 can be used not only to determine the best algorithm for each application but also the length of potentially required buffers, which in this case would be equal to the *maximum* lateness for each accelerator. Evidently, a smaller buffer means lower resource consumption, so obtaining the optimal design in terms of response time and resource consumption comes to an optimal decision in choosing the most fitting scheduling algorithm for the application.

From the modeling point of view, the WCET can be easily added to the system specification to be used for the model analysis and also automate the process to determine both the buffer length and scheduling algorithm.

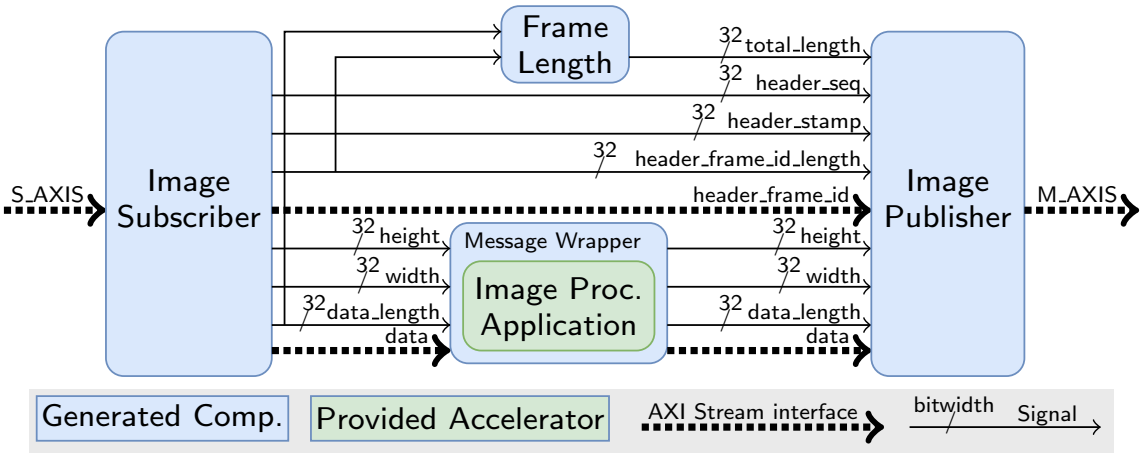


Figure 6.3: Payload of an image publisher dynamically computed

```
1  function MessageLength(Message m)
2    l:=4
3    for each Field f in m do
4      l := l + FieldLength(f)
5    return l
6
7  function FieldLength(Field f)
8    l := 0
9    if f is no-array then
10     l := l + TypeLength(f)
11   else if f is fixed-length array then
12     for i in range(array_length(f)) do
13       l := l + TypeLength(index(f, i))
14   else if f is variable-length array then
15     l := l + 4
16     for i in range(signal(f, length)) do
17       l := l + TypeLength(index(f, i))
18   return l
19
20 function TypeLength(Field f)
21   l := 0
22   if type_of(f) is built-in type then
23     l := size_of(t)
24   else if type_of(f) is message then
25     l := l + 4
26     for each Field s in f do
27       l := l + FieldLength(s)
28   return l
```

Listing 6.5: Computation of message length

6.3.3 Scheduling Transactions between Hardware and Software

Dealing with hybrid hardware/software systems means that the way communication is established between these two has to be addressed. The supported middleware throughout this dissertation is ROS (both versions), so the software part is based on it. Its default scheduling scheme to receive new messages is shown in Figure 6.4a. It consists of a *shared callback queue* for all subscribers. Hence, the callback queue must be read three times (retrieving B) before A can be read in a First In First Out (FIFO) manner. This can cause a message not to be longer usable for a given subscriber. Therefore, a modification to this scheme is proposed (Figure 6.4b) by taking advantage of the use of individual callback queues for each incoming message. This leads to the question of which *spinner thread* should get a hold of the DMA to exchange data between PS and PL. The proposed solution here is inspired by the LRU algorithm, used to manage buffer memories and caches. Unlike a FIFO, the last *spinner*

(a) Default ROS callback scheme

(b) Proposed LRU-based ROS callback scheme

Figure 6.4: ROS scheduling schemes

thread that shares data between PS and PL (reading from its callback queue) is pushed to the end of the priority list (highest priority). This changes dynamically over time, allowing each individual spinner thread to get a hold of the DMA (with the lowest priority) with a maximum delay of N-1 in a round, being N the number of subscribers. A similar analysis as the one shown in Section 4.3 can be followed for the software side. From the software perspective, experiments showed LRU to be good enough. However, the proposed workflow can also be extended to include a similar software evaluation to decide, case by case, the optimal algorithm for both software and hardware counterparts.

The concepts explained until here concerns the transmission from PS to PL. A hardware counterpart is needed to schedule the transactions from PL to PS, which is the reason why multiple algorithms are proposed in Chapter 4. As there are no "one fits all" solutions, having multiple options allows for obtaining the most optimal design for each application. The model-based approach for code generation allows for exchanging these components with minimal effort and, ideally, transparently for the end-user.

Even though the evaluation of the schedulers is fully automatized, the criteria to decide which algorithm will be the most fitted one based on the characteristics specified in the system specification is not yet realized as an optimization algorithm to be integrated into the proposed workflow, which is left for future work.

6.4 FPGA Architectures for Robotics (FAR) Tool

After discussing the toolchain in Section 6.2 and three particular challenges in Section 6.3, this section explains the technical details of the implementation and argues why a model-driven approach is beneficial. Model-driven engineering [131, 90] offers a systematic and domain-oriented development approach using domain-specific models, model transformation, and code generation to create comprehensible and maintainable software. The toolchain shown in Figure 6.2, called FPGA Architectures for Robotics (FAR), has two essential components: the *model analysis* and a set of provided resources and inputs that are used to construct the system using a template engine. In this case, a *logic-less* template approach is used with simple placeholders in the template rather than programmed instructions, simplifying the templates' definition for domain experts. Therefore, all analysis and reasoning must happen *within* the tool.

FAR uses and extends FIRM [9] introduced in Chapter 5, and thus also uses a *grammar-based* modeling approach based on attribute grammars [116]. As opposed to other modeling approaches, grammars describe *trees* rather than models comprised of arbitrarily structured elements. This approach was used to derive all required information to generate the converters for individual messages. For FAR, the middleware-based interfaces generation is also used, and the approach is extended to the generation of the *entire system* [25]. Thus, the analysis must be able to derive all relevant information for the creation from the *system specification* (e.g., in Listing 6.1) and the provided static resources. Attribute grammars are an approach to computing semantic properties of a language (or, in this case, a model) in a declarative and formalized way. In this case, the concept of *higher-order attributes* [120] is used, which additionally allows the computed properties to be entire new artifacts. For this, *relational reference attribute grammars* [33, 121] are employed, which allow efficient linking of tree elements with cross-tree *relations*.

The three challenges identified in Section 6.1 are used to illustrate why such a model-based approach is a necessary and adequate solution to generate hardware/software architectures.

6.4.1 Tailored Information using Intermediate Representations

Since a significant target of the proposed system is to have concise specifications (CH2), most required information to construct a complete system is only included *implicitly*. However, the employed template engine needs all information *explicitly* specified; thus, an analysis with computed attributes on the input model is used. However, doing this transformation in one step is complex and does not allow for reuse (R2) since there are multiple template configurations to be created. Therefore, multiple intermediate representations are employed, i.e., models based on reference attribute grammars obtained using model transformation using higher-order attributes.

One example is an extended system specification model. As suggested in Section 6.3.1, to keep the input specification concise *and* the implementation efficient, signals connecting messages can be filtered using `include` and `exclude` hints. This is a shorthand for the specification of all required signals, which is only possible because the contents and nestings of message types are analyzed. In the full *system specification*, the inclusion hints are expanded to contain a (potentially long) list of all individual fields to be included.

6.4.2 Simplifying Runtime Computation

The computation of the length of the message was already highlighted in Listing 6.5. It consists of two main functions used in a recursive process following the nested structure of a message definition. The first benefit of the chosen approach is that the algorithm can be simplified when considering the intermediate message representation from [9], which no longer contains fixed-length arrays and fewer nested messages, which have been flattened whenever possible. This removes the else branch in lines Line 11 to Line 13 of the algorithm shown in Listing 6.5 and reduces the nesting depth of the function calls. Secondly, since the signal data required in the algorithm are available at different times, function calls have to be inlined depending on the message type. So, again, type analysis is required. Finally, the signals required by the algorithm must be connected, which requires a data flow analysis, which can be performed using the attribute grammar approach[132]. Additionally, optimization can be applied if signals are known at compile time, e.g., when signals are not connected.

6.4.3 Benefits of Model Analysis in the Development Lifecycle

In addition to the analysis previously mentioned and optimization steps, using a model-based, attribute grammar analysis approach allows for further potential analysis improving performance at development time, compile time, and runtime. During development, the construction and verification of the system model can be aided by static analysis, aiding the developer with syntactic and semantic checks, code completion and suggestions, and refactoring support. During compile time, knowledge of the entire system can help with the generation of optimized code beyond the abilities of the FPGA compiler toolchain or

optimizations for better resource utilization. One example of runtime benefits is the use of WCET analysis to ensure real-time guarantees in combination with Listing 6.5 to adapt the scheduling scheme dynamically, knowing the time left for the accelerator.

6.4.4 Details of the Model Analysis

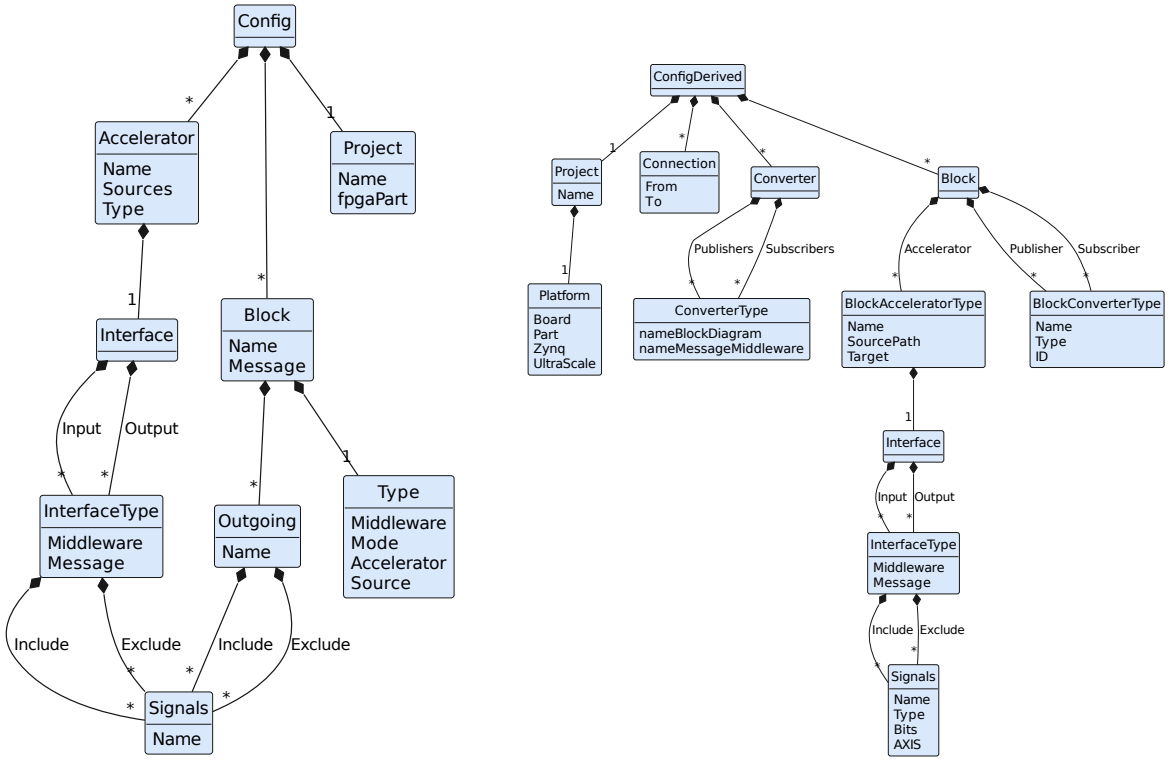
The *system specification* is centered around the user perspective, as it includes characteristics a non-FPGA expert would use to describe the desired system, such as the accelerators or middleware-specific desired interfaces. The first thing to present is the grammar used to represent its structure, shown in Listing 6.6. All the characteristics that can be included as input are *explicitly* defined, as well as their relations. All elements followed with an asterisk * (e.g., Block) represent lists, meaning there can be multiple of them. Its visual representation as a UML class diagram is shown in Figure 6.5a, depicting classes (an object or a set of objects that share a common structure and behavior) and their relations. This is an abstract representation of the *system specification*, and a more specific representation can be obtained by visualizing the AST that is obtained from a concrete *system specification*. It depicts an object (model elements representing instances of a class or of classes) diagram, representing all instances and their relations. Figure 6.6 depicts the AST for the example shown in Figure 6.1, using as input Listing 6.1. It can be seen that it has the three instances of *block*, representing the accelerator and both `message-dependent` components with their respective characteristics (e.g., middleware, type, where it outputs connects to). Having an AST representation like this is beneficial because it allows one to loop through all the objects, simplifying the code generation process.

The *template configuration*, which are the generated artifacts from the model analysis in Figure 6.2, include all low-level and FPGA-related specifications. All this information has to be *derived* from the available information provided by the *systems specification*. As mentioned previously, doing this transformation in one step is complex, so a new grammar (Figure 6.5b) that fits the structure of the *template configuration* is required. This new structure serves as the *intermediate representation* of the extended system's model, containing the required information and fields for the different artifacts that will be generated in later steps of the workflow. The differences between these two grammars can be seen in Figure 6.5, one focusing on the user's perspective and the other one on the low-level details, which are derived from the former one. A snippet of the *template configuration* containing the derived information obtained with the model analysis for the components generated with FIRM is shown in Listing 6.7. Here, there is a further string manipulation based on the *message specification*'s name used to generate the artifact to instantiate these components (TCL script in this case). Three different aspects can be highlighted for the `message-dependend` components. The first one concerns the *type* of converter, either for subscribers (AXIS to message) or publishers (message to AXIS). Then is the message type as defined by the middleware (`nameMW`). These two are the input parameter needed by FIRM to generate the VHDL artifacts. Lastly, `nameBD` represents the *identifier* used in a block design to *instantiate* each `message-dependent` component.

Having the system model represented as an AST is advantageous as it is possible to traverse it easily to obtain all the information that the template engine needs to generate the desired artifacts and do further manipulations, as highlighted in the example above. The analysis performed concerns concretely the transformation from the *system specification* (Figure 6.5a) to the *template configuration* (Figure 6.5b), which is done with attributes.

```
1 Config ::= Project Accelerator* Block*;
2 Project ::= <Name> <fpgaPart>;
3 Accelerator ::= <Name> <Sources> <Type> Interface;
4 Interface ::= Input:InterfaceType* Output:InterfaceType*;
5 InterfaceType ::= <Middleware> <Message> Include:Signals* Exclude:Signals*;
6 Block ::= <Name> Type <DataType> Outgoing*;
7 Type::= <Middleware> <Mode> <Accelerator> <Source>;
8 Outgoing ::= <Name> Include:Signals* Exclude:Signals*;
9 Signals ::= <Name>;
```

Listing 6.6: System specification's grammar



(a) UML representation of the system specification's grammar

(b) UML representation of the derived system specification's grammar

Figure 6.5: UML representations of the system specification ASTs

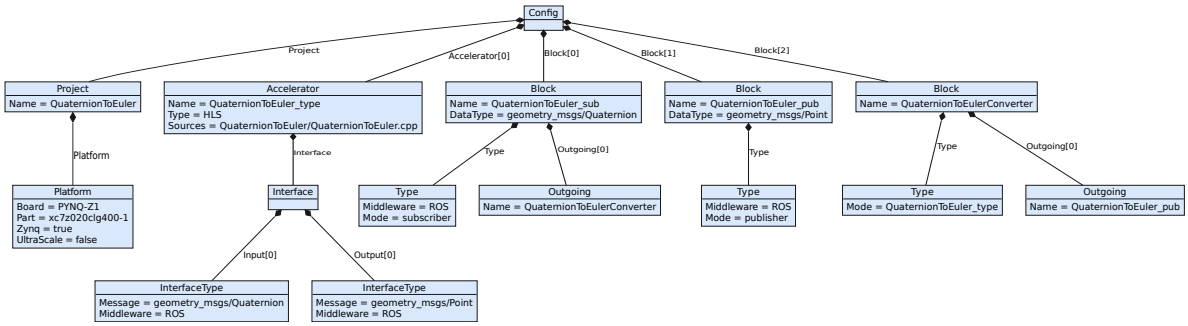


Figure 6.6: Quaternion to Euler's AST.

```
1 Converters:
2   AXIS_to_msg:
3     - nameBD: AXIS_to_geometry_msgs_Quaternion
4       nameMW: geometry_msgs/Quaternion
5   msg_to_AXIS:
6     - nameBD: geometry_msgs_Point_to_AXIS
7       nameMW: geometry_msgs/Point
```

Listing 6.7: Derived configuration file (converters part)

An example of a data-flow attribute to obtain the output interfaces of the specified accelerators is shown in Listing 6.8. The `getInterfaceOutput()` attribute is used for every accelerator in the AST from the *system specification* to obtain the information and build the AST for the *template configuration* with the derived information. Internally, it loops through all the different interface types (output in this case). Line 6 and Line 7 in Listing 6.8 show how the middleware-related information is obtained. The `getVhdlMessage()` attribute (Line 8), taken from FIRM, is used to retrieve all the low-level signals for the *message specification*. Lastly, the signals are *filtered* with `filterIncluded()` (Line 12) and `filterExcluded()` (Line 15) attributes, based on the include and exclude lists from the *system specification*. Here it is also evident the advantages of the use of RAG, allowing to retrieve and manipulate information from complex data structures in a straightforward manner by combining multiple attributes.

Another example of the model analysis is how to derive the connections of all blocks. The *system specification* only defines on a high level which variables from the middleware *message specification* are to be used from one block to another. However, the low-level signals are

```
1 syn ListElement Accelerator.getInterfaceOutput() {
2   ListElement outputs = new ListElement();
3
4   for (InterfaceType otype : getInterface().getOutputList()) {
5       MappingElement out = new MappingElement();
6       out.put("middleware", otype.getMiddleware())
7               .put("message", otype.getMessage());
8       VhdlMessage parsed = containingConfig().getVHDLParsedMsg(otype.
      getMessage()).getMsgToVhdl().getVhdlMessage();
9       if (otype.getIncludeList().getNumChild() == 0 && otype.
      getExcludeList().getNumChild() == 0) {
10          out.put("include", parsed.getListMsgfields(parsed).
      toListElement());
11      } else if (otype.getIncludeList().getNumChild() > 0) {
12          out.put("include", parsed.getListMsgfields(parsed).
      filterIncluded(otype.getIncludeList()).toListElement());
13      }
14      if (otype.getExcludeList().getNumChild() > 0)
15          out.put("include", parsed.getListMsgfields(parsed).
      filterExcluded(otype.getExcludeList()).toListElement());
16      outputs.add(out);
17  }
18  return outputs;
19 }
```

Listing 6.8: Attribute to obtain output interfaces for specified accelerators

needed for the *template configuration* and further on the corresponding artifact. Following a similar approach as the one shown in Listing 6.8, the explicitly defined elements of the *message specification* are obtained and with the help of the `getVhdlMessage()` attribute, the corresponding signals (according to their datatype) are derived. Once that list is generated, further manipulation has to be performed to format the strings accordingly in the *template configuration*. A snippet of this is shown in Listing 6.9. On the one hand, it can be seen (from Line 2 to Line 9) the expected signal. Note that here it is also included which component is the source and which one is the destination, as well as its entity's port name. On the other hand, as the destination block has been defined as HLS, its corresponding signals are deduced, namely `start` and `done`. The clock and reset signals are also included. All these signals are completely transparent for the user, who might not even be aware of them, simplifying the use of FPGAs for non-experts.

6.5 Evaluation

The fulfillment of the *requirements* and how the *challenges* are solved with the *contributions* listed in Section 6.1 are analyzed below, through four different use cases.

6.5.1 Quaternion to Euler

This use case addresses the challenge of obtaining *all* the information (explicit and implicit) from the *system specification* (CH1). Listing 6.1 shows that with only 31 lines of code (without comments and empty spaces for better formatting), the system depicted in Figure 6.1 can be generated and deployed. It can be seen that the input and output signals of the *Quaternion to Euler Converter* have not been individually specified. They have been defined by their message type (Line 15 and Line 18 in Listing 6.1). This means that all the signals that constitute such messages are generated (CH3). Even though they have not been explicitly defined, they are derived by analyzing the message type. The information derived (*template configuration*) also

```
1  Connections:
2    - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/x_out
3      to: QuaternionToEulerConverter/x_in
4    - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/y_out
5      to: QuaternionToEulerConverter/y_in
6    - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/z_out
7      to: QuaternionToEulerConverter/z_in
8    - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/w_out
9      to: QuaternionToEulerConverter/w_in
10   - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/start
11     to: QuaternionToEulerConverter/ap_start
12   - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/done
13     to: QuaternionToEulerConverter/ap_done
14   - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/clk
15     to: processing_system7_0/FCLK_CLK0
16   - from: QuaternionToEuler_sub_AXIS_to_geometry_msgs_Quaternion/rst
17     to: rst_ps7_0_100M/peripheral_aresetn
```

Listing 6.9: Input configuration file (connections part)

includes the integration of the components shown in Figure 6.1 to the PS via DMA, which is where native ROS is running to communicate with external nodes. Additionally, a *wizard* is provided to avoid manually writing the system specification but generate it interactively. This further reduces the possibility of making mistakes in such an error-prone process. The subscriber and publisher take 35 and 28 clock cycles, respectively. The Quaternion to Euler Converter takes 373 clock cycles. Therefore, the interfaces are not an overhead with respect to the time it takes to perform the computation (8%, 6%, and 86%, respectively). Table 6.1 shows the speedup obtained with the Quaternion to Euler conversion in hardware with respect to software running on the PS.

6.5.2 Image Processing

An image processing use case consisting of pipelined functions (i.e., RGB to Grayscale, Downscaling, and Integral computation) was generated. Listing 6.4 shows a snippet of the *system specification* used, defining the interfaces for the accelerators (CH3), which are targeted to be in HLS. It also includes which elements of each interface are connected to where. Table 6.1 shows the execution time of each function. They take images with an input resolution of 1920x1080 (full HD) scaled down to 640x480. A speedup of 12.9x, 18.4x, and 10.2x, respectively, was achieved. In this case, the length of the images (and therefore the resulting AXIS frame) can change. Therefore, the frame length is dynamically computed, as shown in Figure 6.3. The component to compute it is obtained following the algorithm shown in Listing 6.5, and it only consumes 48 LUTs, as it is a purely combinational logic. In this case, the sensor_msgs/Image does not contain nested arrays or messages, so there is no need to buffer any signals to wait for their sizes signals.

6.5.3 Multi-type Messages

A system consisting of multiple *converters* for different types of messages was generated. The different message-specification were chosen to have different lengths and datatypes, namely *sensor_msgs/Image*, *sensor_msgs/LaserScan* and *geometry_msgs/TwistStamped* which results in different transfer times (cf., Section 4.3). Moreover, they also have different frequencies. Each set of converters (one *publisher* and one *subscriber* for each type of message) had a pass-through component in between (considered as the accelerator). This use case aims to evaluate the use of individual callback queues combined with the scheduling as proposed in Section 6.3.3. On the software side, three different callback queues were set. They received

Table 6.1: Execution time of hardware accelerated functions.

Function	Software*[ms]	Hardware+ [ms]	Speedup
Quaternion to Euler	0.012884	0.003730	3.45
Gray Scale Conversion	801.45	62.20	12.9
Scale Down Nearest	381.95	20.73	18.4
Integral	212.22	20.73	10.2
Robotic Arm Kinematics	0.017	0.008	2.12

*Cortex-A9 running at 666 MHz — +HLS IPs running at 100 MHz

three types of ROS messages with different lengths at different frequencies. Depending on the dynamically changing priority list of the LRU scheduler, transactions between PS to PL occurred. On the hardware side, its counterpart (presented in Chapter 4) was used. This use case proved the feasibility of having one callback queue per message, which would correspond to each accelerator. The resource utilization of the hardware implementation has already been analyzed in Figure 4.5, showing a linear growth with respect to the number of inputs (requests from publishers).

6.5.4 Robotic Arm Position Estimation

A system to compute the forward kinematics of a 7 Degrees of Freedom (DoF) robotics arm[3] was generated. This sort of computation becomes relatively complex and proportional to the amount of DoF. This is particularly important when performing motion control by generating a trajectory without colliding with objects. All specifications, inputs, and outputs that describe the robotics arm are defined in a ROS *message specification*. The names used for this evaluation are the ones from that specification. The accelerator is based on the desired and measured joint state values (`q` and `q_d`), and the measured and desired end-effector spatial matrices (`O_T_EE` and `O_T_EE_d`), read from the `franka_msgs/FrankaState` message. The outputs are the pose of each joint as fourteen spatial matrices (`T1` to `T7` and `T1_d` to `T7_d`, based on `q` and `q_d`), and the medium square error (`T_mse`) of the calculated spatial matrices concerning `O_T_EE` and `O_T_EE_d`. The reason why the LoC for the *Generated Artifacts* (Table 6.2) is so large is due to the extend of the franka_msgs/FrankaState message. However, this is not a concern when writing the *system specification* as it only requires including the elements that contain the joint states as the input interface of the HLS accelerator to compute the kinematic equations. Table 6.1 shows a speedup of 2x compared to the software execution, which would be beneficial to perform collision detection by knowing the position of each joint (spatial matrices) as soon as possible.

6.5.5 Manual Vs. Generated Deployment

Table 6.2 compares the LoC that are manually written (or generated interactively via the wizard) of the *system specification* and of all *intermediate artifacts* for all use cases. Even though not all the artifacts would have to be manually written, the ratio between the LoC of the *system specification* and all the *intermediate artifacts* exemplifies the effort needed to deploy each use case manually with respect to the workflow proposed in this work.

The numbers are not an exact representation as they are affected by the message specifications used by each use case and whether *all* signals of that message are used. However, the order of magnitude makes a difference. On the one hand, the first three use cases show one order of magnitude ratio, as the messages used are not very long. On the other hand, the robotics arm use case relies on a quite large and complex message specification, which implies the artifacts for the `message-dependent` components are pretty extensive. Besides, not all elements from that message specification are part of the computation done by the accelerator, but they must be part of the converter because the message is always broadcasted entirely. However, the *system specification* for this use case is the second smallest one

[3] `https://frankaemika.github.io`

Table 6.2: Lines of code of input vs. generated artifacts

Use Case	Input	Generated Artifacts					Generated
	System Specification	Template Configuration	Acc. Wrappers and Scripts	Converters and Scripts	System Components	Combined Artifacts	to Input Ratio
Quaternion to Euler	35	99	22	459	102	682	19.48
Image Processing	83	136	34	692	107	969	11.67
Multi Accelerator System	143	322	34	2320	172	1848	19.91
Robotic Arm	45	1007	22	16540	307	17876	397.24

thanks to the *include* and *exclude* options, resulting in two orders of magnitude difference (in terms of LoC) compared to the total artifacts.

The more complex the project becomes (more accelerators and converters and more complex message specifications), the higher the effort to write every component manually, which does not mean an increment in the effort to write the *system specification*.

6.6 Wizard

The *systems specification* for an architecture like the one shown in Figure 6.1 can be written manually as shown in Listing 6.1. The process needs to be meticulous about the special format and indentation required by the YAML format. However, for larger and more complex systems with multiple accelerators involved, writing it manually becomes error-prone. Therefore, an interactive wizard has been included in FAR to aid the user in generating the *system specification*. It is based on the grammar shown in Listing 6.6. Hence, interactively, it traverses it and builds dynamically through the command line a resulting AST. Once all the elements and characteristics of the desired system have been entered, a YAML file is generated, which can be used as the input for FAR shown in Figure 6.2. The block diagram in Figure 6.7 shows an overview of the procedure, which prompts the corresponding elements according to the different elements to be entered. For example, the *type* of the accelerator suggests the three currently supported options, namely VHDL, HLS, or software. One last characteristic of the wizard is that it not only helps to create a new *system specification* from scratch, but it is possible to load an existing one for further work from within the wizard.

Implementing such a command line tool is not complex as it is a matter of traversing the AST correctly (knowing whether a branch represents a list or not, based on its grammar), which is quite simple thanks to relying on RAGs, as shown previously. However, a wizard via command line might be a familiar tool for a computer scientist, but a graphical approach might be a more general solution, which is left for future work.

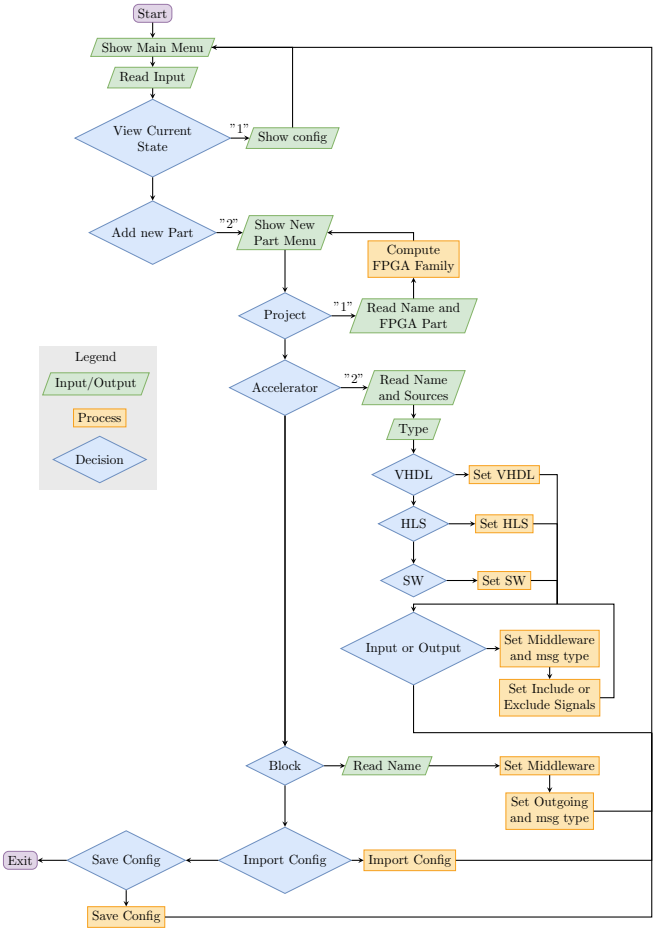


Figure 6.7: Workflow of interactive tool to create a system specification interactively.

6.7 Adaptability and Extendability

Even though the evaluation in Section 4.3 showed that the proposed architecture (Figure 3.1) can be scaled up with the *Manager* thanks to the proposed schedulers in Chapter 4, there could be cases where other components are a better option. For example, NoCs have been proposed [133] and studied for large scalable systems [134, 135, 136], so they are a good fit to replace the *Manager*. Having a NoC in the system means that the *Manager* is replaced, and therefore, there would be no need for the proposed schedulers.

Replacing or adding components requires some changes to adapt mainly FAR (Section 6.4) to new requirements and extend it to fulfill them by generating these new components and integrating them into the existing architecture shown before. Considering the case where a NoC is the chosen component to replace the *Manager* with, the first thing to modify is the grammar for the *derived system's specification* (Figure 6.5b), which needs to reflect the new component of the system that is to be generated. This grammar has to be extended to contain all the derived information via model analysis regarding the characteristics of the NoC to generate its components (e.g., number of nodes, topology, routing algorithm) that are most fitted for the specified system. Note that the grammar for the *system's specification* (Figure 6.5a), which is used for FAR's input (written by the user), does not need to be modified

as this extension of the system is *transparent* to the end-user.

Figure 6.8 shows the class diagram of the extended grammar for the system to include a NoC. The nodes in the resulting AST corresponding to the extended grammar represent the characteristics of the NoC. The topology is the first thing to define, which will dictate the routing algorithm for its routers. Each router will have an (*X*, *Y*) coordinate to define its position if a *mesh topology* is chosen, for example. Besides, connection ports to other routers and a PE for each are also defined. Lastly, the source of the PE is expected, similarly to the source of the *accelerators* as explained previously.

Once the grammar has been extended, the details of the system have to be derived with model analysis. To do this, some attributes can be re-used. For example, to count how many accelerators (which for the NoC determines the total number of routers) are specified. Each accelerator acts as the PE for each router. Then this information will then be the input for *a new attribute* to determine the dimension (*X*, *Y*) of the NoC. Lastly, new attributes are required to analyze each accelerator based on potential given characteristics (i.e., WCET) to map them onto the NoC. This can also lead to specific NoC-based attributes to perform DSE to obtain the optimal position for each accelerator in the NoC. Once the position of each accelerator in the given topology has been established, the connections to its router follow. In this case, as the target of this dissertation is robotics applications, the interfaces will be generated with FIRM,

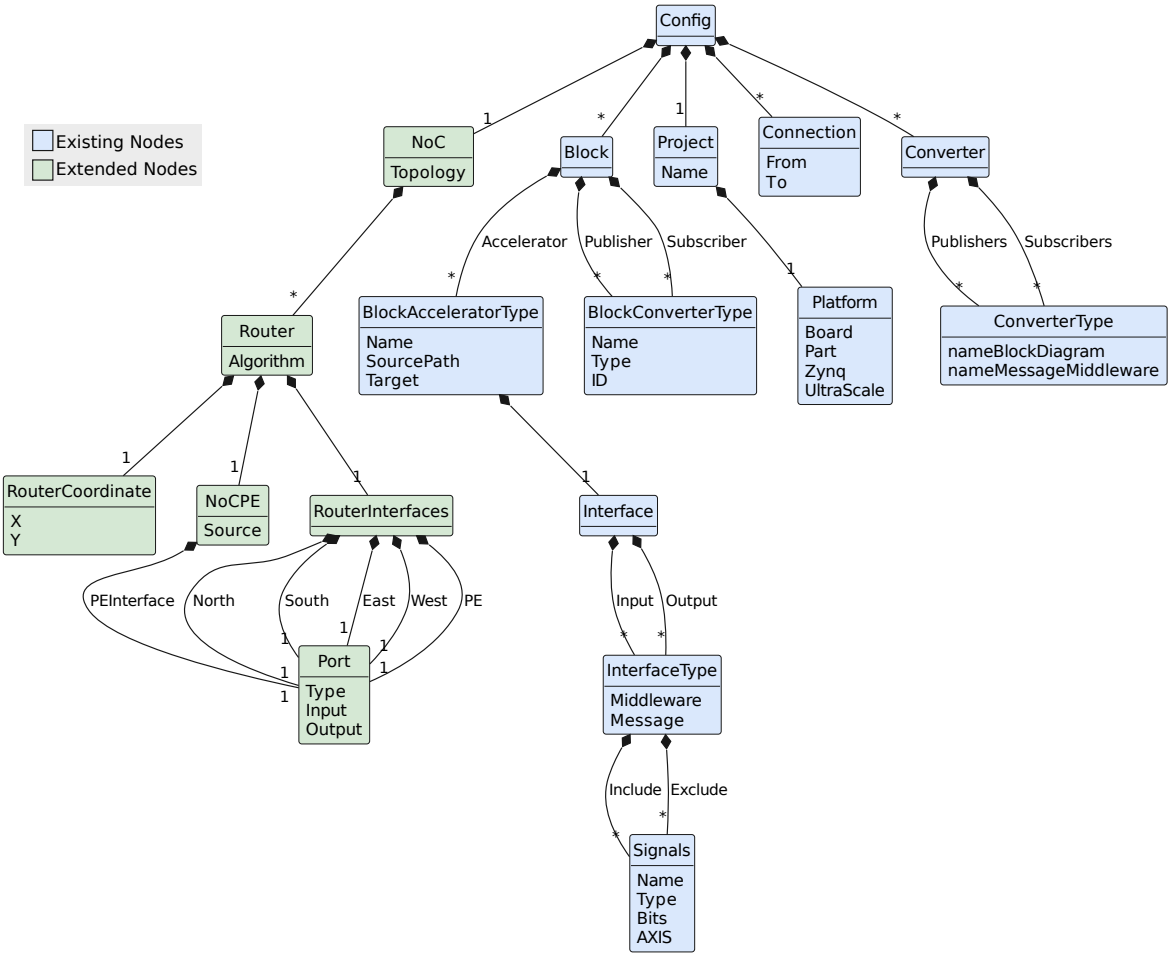


Figure 6.8: Adapted grammar including a Network-on-Chip.

following the methodology presented earlier in this chapter. However, this is not mandatory, as the methodology described here can also be used for non-robotics applications or other middlewares besides ROS. In those cases, FIRM will have to be adapted.

Once the model analysis has been done, all the information to deploy the NoC and the position of the accelerators have been determined. This information is used to extend the template configuration. Therefore, new templates are required to use this derived information to build the component as a hardware IP for the NoC. The advantage of generating a *custom* NoC is that the port's entity can be generated in terms of the number of accelerators (and routers) for each application. Then a similar outcome to the one shown in Listing 6.9 follows, which is to deploy this new component to replace the *Manager*. Concerning the automatic deployment, the instantiation of the *Manager* has to be replaced to instantiate the autogenerated NoC in the script that creates the project with all the components defined to constitute a robotics system. There is one detail to keep in mind. The *Manager* has been designed to have one master and one slave AXIS port to connect them to the DMA. In this example case of extending the system to contain a NoC, the first approach is to use any of its nodes for communicating with the DMA. Traditionally in a mesh topology, each router has four ports to communicate with other routers (north, south, east and west). The advantages in this case relying on the MDE technique is that all edge routers which do not need all communication ports do not need to leave unconnected signals but rather *generate* tailored routers depending on their coordinate (X, Y) dictating which ports they need. It is left for future work to explore how to improve this, either at the algorithm level for the routers or the mapping part.

6.8 Summary

This chapter focuses on the automatic generation of holistic hardware/software architectures for FPGA-based robotics systems with the FAR tool. It uses and extends FIRM, presented in Chapter 5, meaning that robotics middlewares and their specifications are a central part of the expected outcome. One main challenge is reducing the effort to realize said architectures. Thus, the input of the proposed workflow used to specify the system must be concise yet expressive enough that it explicitly contains all characteristics of the desired systems or implicit in a way that they can be deduced. Consequently, the core of this chapter focuses on the data-type and data-flow analysis required to obtain all the information to generate the different artifacts for every architecture.

Following the MDE approach, details about the grammar used for the *system specification* as well as for the derived system's model are given to understand further the advantages of RAGs for hardware-centric code generation and how the analysis is done via attributes. An evaluation is performed with heterogeneous use cases to highlight the benefits of the proposed techniques, showing in terms of LoC the differences in orders of magnitude that would represent deploying these large and complex systems manually compared to using FIRM in combination with FAR.

Two points are left for future work. On the one hand, one challenge concerning the data transmission between PS and PL with ROS involved is addressed. The solution based on multiple callback queues is proposed, which can be combined with the different schedulers proposed in Chapter 3. However, the integration of an algorithm to select which scheduling algorithm is the most fitted one for every system to be generated is left for future work. On

the other hand, the *system specification* written in YAML requires following strict rules related to the language it is written. This is no problem when the system does not include many `message-dependent` components or accelerators. However, it becomes pretty error-prone, which would decrement the benefits gained by generating the rest of the system automatically if the input of the proposed tool FAR introduces complexity into the process. Even though a command-line tool is included in FAR, a Graphical User Interface (GUI)-based generation of *system specifications* is left for future work to research aspects on the grammar-based generation of system specifications from a GUI and using the features of attribute grammars (static analysis) to aid the user in the process.

7 Conclusion

The current trend towards increasing heterogeneity in modern robotics platforms and applications has highlighted the need for computational solutions that can meet the strict real-time constraints imposed by these systems. The standard embedded CPUs typically used in robots have limitations in processing the vast amounts of data and complex algorithms that are increasingly common in these systems. As a result, there is a considerable need to find alternative solutions for PEs that can fulfill these real-time demands effectively.

On the one hand, GPUs have been widely adopted as a solution for PEs in many applications, including robotics. However, like any technology, GPUs have both benefits and drawbacks that must be considered when evaluating their suitability for a particular application. GPUs have a more straightforward programming model compared to other options, such as FPGAs. This makes GPUs more accessible for software developers and reduces the time and resources required to develop and implement solutions. In terms of high performance, GPUs are designed to perform parallel computations on large amounts of data, making them ideal for applications that require high-performance computation. In addition, GPUs often have more memory bandwidth and computing power than traditional CPUs, allowing for more efficient processing of large amounts of data. Lastly, GPUs are widely used in many applications, including computer graphics and scientific computing, making them a well-established and widely supported technology. This provides a large pool of expertise and resources that can be leveraged to develop and implement solutions. In summary, GPUs have many benefits that make them a suitable solution for some robotics applications. However, their limitations and drawbacks must also be considered when evaluating their suitability for a particular application. GPUs are designed for specific functionalities, making it difficult to adapt to the unique and constantly evolving requirements of robotics systems. Additionally, they do not provide the level of customization and adaptability needed in robotics applications. As a result, GPUs may not be the best solution for processing elements in all robotics application cases, especially when dealing with complex algorithms and large amounts of data that require real-time processing. It is essential to carefully consider the application's requirements and choose a solution capable of meeting those requirements while balancing the benefits and drawbacks of the available options.

On the other hand, FPGAs are an alternative solution for PEs in robotics and other applications alike. As with GPUs, FPGAs have their own set of benefits and drawbacks that must be considered when evaluating their suitability for a particular application. They are highly versatile, providing almost no limitations in terms of functionality. This makes them ideal for applications that require advanced capabilities and specialized functions. FPGAs are capable of computing multiple things (i.e., processing data from sensors, algorithms) concurrently, making them well-suited for applications that require real-time processing and low latency.

Lastly, they are highly efficient, with low power consumption, making them well-suited for applications that are constrained by size and power requirements, such as mobile and embedded systems. However, FPGAs have limitations in terms of programmability compared to other options, such as GPUs. This can result in increased development time. Additionally, special low-level and hardware-related skills knowledge are required to design FPGA-based systems that might not be in the skill set of all development teams, especially in robotics. Hence, this makes it more challenging to develop and implement solutions. Lastly, the debugging and verification of FPGA-based designs can be time-consuming and complex, requiring specialized tools and techniques. In summary, FPGAs offer a high degree of versatility and processing capability, making them a suitable solution for many robotics applications. However, their programmability and specialized skill requirements can make them more challenging to implement and maintain than other options, such as GPUs. Therefore, it is important to consider the application's requirements carefully, as well as the available resources and expertise within the development team when evaluating the suitability of FPGAs for a particular application.

Considering this, FPGAs have been proposed as a suitable solution due to their capability to handle complex algorithms and perform concurrent computing. They are the chosen PEs in this dissertation. Their use in robotics, however, presents particular technical challenges. The primary obstacle is the difficulty in programming them, which limits their widespread adoption by the robotics community. Despite this, using FPGAs can offer significant benefits in robotics, as long as they do not make the existing robotics workflow even harder. Hence, this dissertation has undertaken further research to address the challenges associated with programming FPGAs and explore their potential benefits in robotics applications. Ultimately, the successful implementation of FPGAs in robotics can significantly advance the state-of-the-art in this field, enabling the development of more advanced and capable robotic systems.

The development of robotic platforms involves expertise from multiple fields, including hardware, software, and control systems. Integrating diverse computing systems and components into these platforms further complicates their design and operation. Specialists in each field must focus on their area of expertise to achieve optimal results while complementing each other in the development process. However, the challenges in integrating these complex and heterogeneous systems into a unified platform still remain. It is essential to provide designers with simple and efficient tools that allow them to concentrate on their areas of expertise to overcome these challenges.

A component-oriented approach is beneficial in designing FPGA-based systems for robotics applications because it enables modularity and separation of concerns. This approach allows different system components to be developed and tested independently, reducing the complexity of the overall system and facilitating debugging and maintenance. The modular design also enables easy replacement and upgrade of individual components without affecting the rest of the system. Furthermore, separating concerns fosters specialization, allowing each component to focus on a specific task, leading to improved performance and increased reusability. In short, the component-oriented approach leads to more organized, scalable, and maintainable systems.

Therefore, this dissertation proposes a component-oriented approach to simplify FPGA-based design in robotics, to make the process accessible and efficient, and preserve the versatility and real-time processing capabilities of FPGAs. This approach enables easy integration into a system or architecture by utilizing code-generation based on MDE for obtaining

components from simple system specifications to automatically obtain and deploy a full FPGA-based robotics application. The methodology is application-independent, and generating new components and systems for further applications is straightforward. Additionally, it closes the gap in the current state-of-the-art as there have been partial contributions addressing the use of FPGAs for specific robotics applications. However, a comprehensive examination of their integration into the field of robotics and holistically considering FPGAs have been lacking in the literature.

The proposed component-oriented approach aims to simplify the design process of FPGA-based designs for robotics applications. The components are designed to depend on each other, as their functionalities are complementary, allowing for a more efficient design process. An architecture is proposed to serve as the base for all generated systems. It is crucial for successful Hardware/Software Co-Designs as it includes the essential components needed to exchange data between software and hardware components seamlessly. To ensure the end solution is functional, the base architecture must be versatile and able to accommodate various applications, no matter their complexity. Moreover, the architecture is designed to be scalable and able to adapt to the growing needs of complex systems without requiring excessive effort for generation and deployment. This property makes it a solid foundation for integrating FPGAs into robotics systems, especially considering that these systems often consist of multiple, potentially complex components. The architecture is designed to be highly flexible, making it possible to incorporate new components and update existing ones as needed easily. The modular design also enables the reuse of components across different designs, further simplifying the design process and reducing the time and effort required for design implementation. One key component in the base architecture is the scheduler, which handles data transactions between software and hardware components. Several algorithms implemented as hardware components are proposed and tested for scalability. This evaluation also helps to analyze each system specification to asses which of these schedulers would fit best for each application.

Traditional robotics systems are composed of various software applications that are connected together through a middleware layer. The middleware serves as the intermediary between the different software components, facilitating the exchange of data between them by following pre-defined message protocols. The pre-defined specifications for transmitting data between software and hardware components must be mapped into additional hardware components to achieve the seamless integration of FPGAs into traditional robotics systems. The mapping process must ensure that data exchange between hardware and software components is seamless and effortless. The generation of these components must also be achieved without affecting the design flow of the systems. The entire process must be transparent to the designer. Then, these components act as hardware interfaces that are generated based on the pre-defined (message) specifications from robotics middlewares. They translate incoming or outgoing messages from/to the accelerators (where the computation is performed). They are connected to them using either given parts of the message specification (i.e., a set of signals representing variables) or a standard streaming interface such as the AXIS for arrays, depending on the needs of the accelerator. Several steps are required to obtain these components, which are performed by the proposed tool FIRM based on the MDE technique, in which models are transformed in iterations. In this case, relying on models benefits the design of hardware components based on message specification. The specifications can be complex, and manually designing the state machines for the hardware interfaces can be cumbersome, particularly with complex specifications. Additionally, a complex robotics system comprises several parts with different message specifications,

which benefits from automatic code generation. An extensive evaluation is performed to validate the correctness of the generated logic for the components obtained by FIRM, as it must be able to support and generate hardware components for any off-the-shelf or custom message specification. The evaluation process ensures this. For this dissertation, the most popular robotics middleware ROS was used in both current versions, and the evaluation included all available public message specifications. The MDE approach simplified the process of extending FIRM when ROS2 was added to the tool. The analysis in terms of the diversity, complexity, and size of these messages proved to be valid to confirm the extensive support of ROS1 and ROS2 by FIRM.

Once all the components are available, they need to be connected all together. In order to do so, an understanding of the expected system is needed. Besides, the components of the base architecture are generated depending on the specific needs. For this, data-type and data-flow analysis is performed on the system specification to derive which are the interface components that need to be generated with FIRM, which accelerators are present in the design, and how all of them interact among each other. Hence, FIRM has been incorporated into the FAR toolchain, which only takes a simple system specification as input. Its purpose is to automatically deploy the entire system, making the design process more efficient and straightforward. FAR system automates the deployment of the design, reducing the time and effort required for manual deployment. It also provides a convenient way to update and maintain the system, making it easier to keep the design up to date and ensure optimal performance. Hence, FAR's output is the entire generation and deployment process of an FPGA-based robotics application from a concise description of the expected system.

In conclusion, the component-oriented approach proposed in this dissertation provides a practical and scientifically rigorous solution for the design and implementation of FPGA-based designs for robotics applications. An overview of this dissertation is shown in Figure 7.1. The modular architecture, the hardware interfaces generator FIRM, and the toolchain FAR provide a comprehensive design process that enables the development of complex FPGA-based designs in a more straightforward and efficient manner. The component-oriented approach has the potential to advance the state-of-the-art in FPGA-based designs significantly for robotics applications and to promote their wider adoption and use by specialists without much FPGA knowledge.

Future work

The focus of this dissertation was to investigate the seamless integration of FPGAs into robotics systems. This integration was achieved by developing the FIRM tool and the FAR toolchain. The results of the study indicate that a significant advancement has been made in this direction. However, there are still opportunities for further improvement and open topics that can be addressed in future work. These directions include: (1) replacing the scheduler and modeling a NoC as part of the modular architecture, (2) simplifying the design process of interface templates, and (3) modeling the behavior of accelerators with a tool, also capable of optimizing resource utilization and of being included as part of FAR. By addressing these points, future work has the potential to enhance the integration of FPGAs into robotics systems, resulting in even more efficient and effective solutions.

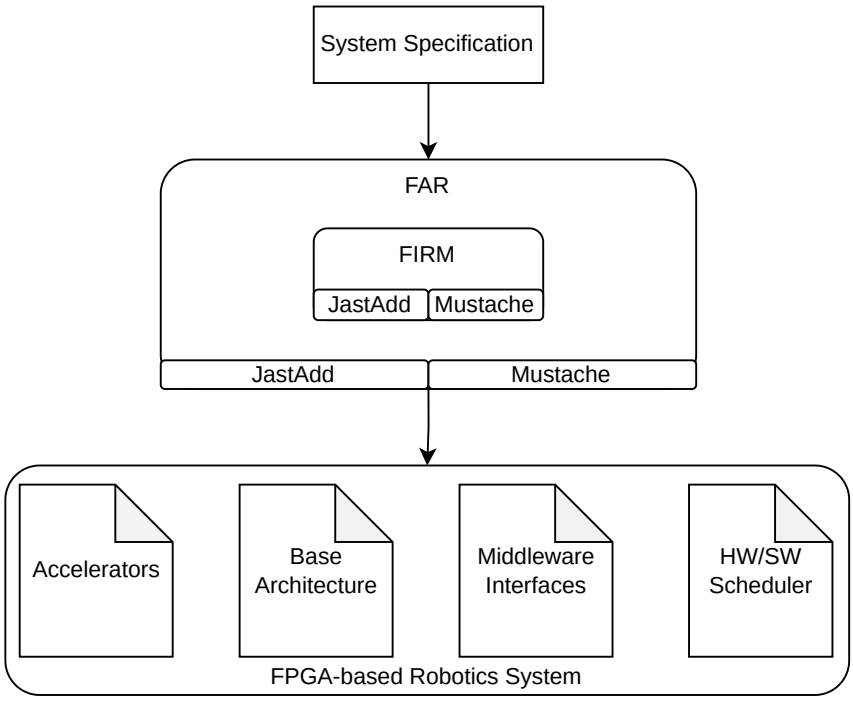


Figure 7.1: Dissertation overview

Analysis and Code Generation for NoCs: Hardware/Software Co-Design was proposed in this dissertation with a component that integrates a scheduler (with several algorithms implemented to choose from), which might be replaced with a NoC. The goal is to take advantage of model analysis to generate the input and output ports of the NoC in a tailored way for each specific application instead of a generic solution which can include unused resources when deployed to the FPGA. This can also impact the routing algorithm, which can be modeled accordingly. The research can explore how the code can be tailored for each specific router in each coordinate of the NoC and how the links between routers are affected. Similarly to the schedulers proposed in this dissertation, it can be evaluated which NoC topology and routing algorithm would fit best for each application.

Model-based Templates Generation from Specifications: The process of designing the templates for the interface components is time-consuming and arduous and can be improved. Currently, the templates are created using message examples, which can become difficult to debug as they grow larger, leading to increased development time. To simplify this process, future work could focus on *generating* templates based on message specification rules and protocols. The research would explore what modeling techniques can be used and how they can be integrated into the existing tools, FIRM and toolchain FAR. Even though a new set of templates was created for ROS2 by extending the ones for ROS1, the process was lengthy due to differences in the rules used for serializing/deserializing the messages to optimize memory usage. This future work would not only simplify the process of creating templates but also make it easier to extend the tools to new communication specifications, like protobuf, which requires a different set of templates due to differences in the rules used for serializing/deserializing messages.

Model-base Design of Hardware Accelerators: The current limitation of FAR is the requirement of pre-existing accelerators. In their absence, FAR generates placeholder wrappers derived from data-type and data-flow analysis of the system specification. Future research may address this limitation by focusing on modeling accelerators from a behavioral perspective while considering resource utilization restrictions. The ultimate goal is to integrate these accelerators into the design process of FAR, leading to the optimization of resource utilization and making it beneficial for low-power applications.

Bibliography

[1] Guang-Zhong Yang et al. "The Grand Challenges of Science Robotics". *Science robotics* 3.14 (2018), eaar7650. DOI: 10.1126/scirobotics.aar7650.

[2] Arkadeep Kumar. "Methods and Materials for Smart Manufacturing: Additive Manufacturing, Internet of Things, Flexible Sensors and Soft Robotics". *Manufacturing Letters* 15 (2018), pages 122–125. DOI: 10.1016/j.mfglet.2017.12.014.

[3] Ana Correia Simões, António Lucas Soares, and Ana Cristina Barros. "Factors Influencing the Intention of Managers to Adopt Collaborative Robots (Cobots) in Manufacturing Organizations". *Engineering and Technology Management* 57 (2020), page 101574. DOI: 10.1016/j.jengtecman.2020.101574.

[4] Fernando Soto and Robert Chrostowski. "Frontiers of Medical Micro/Nanorobotics: In Vivo Applications and Commercialization Perspectives Toward Clinical Uses". *Frontiers in Bioengineering and Biotechnology* 6 (2018), page 170. DOI: 10.3389/fbioe.2018.00170.

[5] UM Rao Mogili and BBVL Deepak. "Review on Application of Drone Systems in Precision Agriculture". *Procedia Computer Science* 133 (2018), pages 502–509. DOI: 10.1016/j.procs.2018.07.063.

[6] Mary B Alatise and Gerhard P Hancke. "A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods". *IEEE Access* 8 (2020), pages 39830–39846. DOI: 10.1109/ACCESS.2020.2975643.

[7] Kaveh Azadeh, René De Koster, and Debjit Roy. "Robotized and Automated Warehouse Systems: Review and Recent Developments". *Transportation Science* 53.4 (2019), pages 917–945. DOI: 10.2139/ssrn.2977779.

[8] Abadi Martin et al. "TensorFlow: A System for Large-Scale Machine Learning". *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pages 265–283. DOI: 10.5281/zenodo.4724125.

[9] Ariel Podlubne, Johannes Mey, René Schöne, Uwe Aßmann, and Diana Göhringer. "Model-Based Approach for Automatic Generation of Hardware Architectures for Robotics". *IEEE Access* 9 (2021), pages 140921–140937. DOI: 10.1109/ACCESS.2021.3119061.

[10] Yasuhiro Nitta, Sou Tamura, and Hideki Takase. "A Study on Introducing FPGA to ROS Based Autonomous Driving System". *International Conference on Field Programmable Technology (FPT)*. 2018. DOI: 10.1109/FPT.2018.00090.

[11] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pages 203–215. DOI: 10.1109/TCAD.2006.884574.

[12] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC". *International Conference on Field Programmable Logic and Applications (FPL)*. 2016. DOI: 10.1109/FPL.2016.7577314.

[13] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. "Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA". *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2017), pages 35–47. DOI: 10.1109/TCAD.2017.2705069.

[14] Lester Kalms, Ariel Podlubne, and Diana Göhringer. "HiFlipVX: An Open Source High-Level Synthesis FPGA Library for Image Processing". *International Symposium on Applied Reconfigurable Computing (ARC)*. 2019. DOI: 10.1007/978-3-030-17227-5_12.

[15] Jiantao Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". *International Symposium on Field Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2016, pages 26–35. DOI: 10.1145/2847263.2847265.

[16] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an Open-Source Robot Operating System". *International Conference on Robotics and Automation (ICRA) - Workshop on Open Source Software*. Kobe, Japan. 2009, page 5.

[17] Lester Kalms, Ariel Podlubne, and Diana Göhringer. "HiFlipVX: An open source high-level synthesis fpga library for image processing". *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer. 2019, pages 149–164. DOI: 10.1007/978-3-030-17227-5_12.

[18] Lester Kalms and Diana Göhringer. "Exploration of OpenCL for FPGAs Using SDAccel and Comparison to GPUs and Multicore CPUs". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2017. DOI: 10.23919/FPL.2017.8056847.

[19] Ariel Podlubne and Diana Göhringer. "Reconfigurable Computing Systems as Component Oriented Designs for Robotics". *International Conference on Field Programmable Logic and Applications (FPL)*. 2021, pages 1–4. DOI: 10.1109/FPL53798.2021.00052.

[20] Ariel Podlubne and Diana Göhringer. "Modeling FPGA-based Architectures for Robotics Systems". *International Conference on Field Programmable Technology (FPT)*. IEEE, 2022, pages 1–4. DOI: 10.1109/ICFPT56656.2022.9974412.

[21] Ariel Podlubne and Diana Göhringer. "FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs". *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2019, pages 1–5. DOI: 10.1109/ReConFig48160.2019.8994719.

[22] F. J. Furrer. *Future-Proof Software-Systems*. Springer, 2019, pages 107–108. DOI: 10.1007/978-3-658-19938-8_4.

[23] Uwe Aßmann et al. *Tactile Internet with Human-in-the-Loop*. Elsevier, 2021. Chapter U2: "Human-robot Cohabitation in Industry", pages 41–73. DOI: 10.1016/B978-0-12-821343-8.00013-7.

[24] Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis. "Modeling for Verification". *Handbook of Model Checking*. Edited by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pages 75–105. DOI: 10.1007/978-3-319-10575-8_3.

[25] Ariel Podlubne, Johannes Mey, Sergio Pertuz, Uwe Aßmann, and Diana Göhringer. "Model-based Generation of Hardware/Software Architectures for Robotics Systems". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2022, pages 1–7. DOI: 10.1109/FPL57034.2022.00034.

[26] Ariel Podlubne, Johannes Mey, Andreas Andreou, Sergio Pertuz, Uwe Aßmann, and Diana Göhringer. "Model-based Generation of Hardware/Software Architectures with Hybrid Schedulers for Robotics Systems". *IEEE Transactions on Computers* (2023). DOI: 10.1109/TC.2023.3323804.

[27] Ariel Podlubne and Diana Göhringer. "A survey on Adaptive and Parallel Computing in Robotics: Modelling, Methods and Applications". *IEEE Access* 11 (2023), pages 53830–53849. DOI: 10.1109/ACCESS.2023.3281190.

[28] Schmidt. "Model-Driven Engineering". *Computer Society, Computer* 39.2 (2006), pages 25–31. DOI: 10.1109/MC.2006.58.

[29] Felleisen. "On the Expressive Power of Programming Languages". *Science of Computer Programming* 17.1-3 (1991), pages 35–75. DOI: 10.1007/3-540-52592-0_60.

[30] Andrea Suardi, Eric C Kerrigan, and George A Constantinides. "Fast FPGA Prototyping Toolbox for Embedded Optimization". *International Conference on European Control Conference (ECC)*. IEEE. 2015, pages 2589–2594. DOI: 10.1109/ECC.2015.7330928.

[31] Alberto Rodrigues da Silva. "Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model". *Computer Languages, Systems and Structures* 43 (2015), pages 139–155. DOI: 10.1016/j.cl.2015.06.001.

[32] Amelie Flatt, Arne Langner, and Olof Leps. *Model-Driven Development of Akoma Ntoso Application Profiles: A Conceptual Framework for Model-Based Generation of Xml Subschemas*. Springer Nature, 2023. DOI: 10.1007/978-3-031-14132-4.

[33] Görel Hedin. "Reference Attributed Grammars". *Informatica (Slovenia)* 24.3 (2000), pages 301–317.

[34] Sven Karol. "An Introduction to Attribute Grammars". *Department of Computer Science. Technische Universitat Dresden, Germany* (2006).

[35] Görel Hedin and Eva Magnusson. "JastAdd: an Aspect-Oriented Compiler Construction System". *Science of Computer Programming* 47.1 (2003), pages 37–58. DOI: 10.1016/S0167-6423(02)00109-0.

[36] Torbjörn Ekman and Görel Hedin. "The JastAdd Extensible Java Compiler". *International Conference on Object-Oriented Programming Systems, Languages and Applications*. 2007, pages 1–18. DOI: 10.1145/1297027.1297029.

[37] Görel Hedin. "An Introductory Tutorial on JastAdd Attribute Grammars". *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2009, pages 166–200.

[38] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. "A Survey of FPGA-Based Robotic Computing". *Circuits and Systems Magazine* 21.2 (2021), pages 48–74. DOI: 10.1109/MCAS.2021.3071609.

[39] Minxi Jin and Tsutomu Maruyama. "Fast and Accurate Stereo Vision System on FPGA". *Transactions on Reconfigurable Technology and Systems (TRETS)* 7.1 (2014), pages 1–24. DOI: 10.1145/2567659.

[40] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. "Real-Time High-Quality Stereo Vision System in FPGA". *Transactions on Circuits and Systems for Video Technology* 25.10 (2015), pages 1696–1708. DOI: 10.1109/TCSVT.2015.2397196.

[41] Oscar Rahnama, Tommaso Cavalleri, Stuart Golodetz, Simon Walker, and Philip Torr. "R3SGM: Real-Time Raster-Respecting Semi-Global Matching for Power-Constrained Systems". *International Conference on Field Programmable Technology (FPT)*. IEEE. 2018, pages 102–109. DOI: 10.1109/FPT.2018.00025.

[42] Michal C Malin et al. "The Mars Science Laboratory (MSL) Mast Cameras and Descent Imager: Investigation and Instrument Descriptions". *Earth and Space Science* 4.8 (2017), pages 506–539. DOI: 10.1002/2016EA000252.

[43] Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. "Real-Time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability Trade-OFFs of the Altera OpenCL SDK". *International Conference on Field Programmable Technology (FPT)*. IEEE. 2014, pages 326–329. DOI: 10.1109/FPT.2014.7082810.

[44] Mohamed Abouzahir, Abdelhafid Elouardi, Rachid Latif, Samir Bouaziz, and Abdelouahed Tajer. "Embedding SLAM Algorithms: Has It Come of Age?" *Robotics and Autonomous Systems* 100 (2018), pages 14–26. DOI: 10.1016/j.robot.2017.10.019.

[45] Konstantinos Boikos and Christos-Savvas Bouganis. "Semi-Dense SLAM on an FPGA Soc". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pages 1–4. DOI: 10.1109/FPL.2016.7577365.

[46] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. "The Microarchitecture of a Real-Time Robot Motion Planning Accelerator". *International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pages 1–12. DOI: 10.1109/MICRO.2016.7783748.

[47] Uday Bondhugula, Ananth Devulapalli, James Dinan, Joseph Fernando, Pete Wyckoff, Eric Stahlberg, and P Sadayappan. "Hardware/Software Integration for FPGA-Based All-Pairs Shortest-Paths". *International Symposium on Field Programmable Custom Computing Machines (FCCM)*. IEEE. 2006, pages 152–164. DOI: 10.1109/FCCM.2006.48.

[48] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. "GPU Computing". *IEEE* 96.5 (2008), pages 879–899. DOI: 10.1109/JPROC.2008.917757.

[49] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. "Performance Comparison of FPGA, GPU and CPU in Image Processing". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2009, pages 126–131. DOI: 10.1109/FPL.2009.5272532.

[50] David H Jones, Adam Powell, Christos-Savvas Bouganis, and Peter YK Cheung. "GPU Versus FPGA for High Productivity Computing". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2010, pages 119–124. DOI: 10.1109/FPL.2010.32.

[51] Sparsh Mittal. "A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform". *Systems Architecture* 97 (2019), pages 428–442. DOI: 10.1016/j.sysarc.2019.01.011.

[52] Gopalakrishna Hegde and Nachiket Kapre. “CaffePresso: Accelerating Convolutional Networks on Embedded SoCs”. *Transactions on Embedded Computing Systems (TECS)* 17.1 (2017), pages 1–26. DOI: 10.1145/3105925.

[53] John M Pierre. “Spatio-Temporal Deep Learning for Robotic Visuomotor Control”. *International Conference on Control, Automation and Robotics (ICCAR)*. IEEE. 2018, pages 94–103. DOI: 10.1109/ICCAR.2018.8384651.

[54] Ze Wang, Weiqiang Ren, and Qiang Qiu. “Lanenet: Real-Time Lane Detection Networks for Autonomous Driving”. *arXiv preprint arXiv:1807.01726* (2018). DOI: 10.48550/arXiv.1807.01726.

[55] Nitin J Sanket, Chahat Deep Singh, Kanishka Ganguly, Cornelia Fermüller, and Yiannis Aloimonos. “Gapflyt: Active Vision Based Minimalist Structure-Less Gap Detection for Quadrotor Flight”. *Robotics and Automation Letters* 3.4 (2018), pages 2799–2806. DOI: 10.1109/LRA.2018.2843445.

[56] Ratnesh Madaan, Daniel Maturana, and Sebastian Scherer. “Wire Detection using Synthetic Data and Dilated Convolutional Networks for Unmanned Aerial Vehicles”. *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pages 3487–3494. DOI: 10.1109/IROS.2017.8206190.

[57] Nasrin Attaran, Abhilash Puranik, Justin Brooks, and Tinoosh Mohsenin. “Embedded Low-Power Processor for Personalized Stress Detection”. *Transactions on Circuits and Systems II: Express Briefs* 65.12 (2018), pages 2032–2036. DOI: 10.1109/TCSII.2018.2799821.

[58] Tahmid Abtahi, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. “Accelerating Convolutional Neural Network with FFT on Embedded Hardware”. *Transactions on Very Large Scale Integration (VLSI) Systems* 26.9 (2018), pages 1737–1749. DOI: 10.1109/TVLSI.2018.2825145.

[59] Ali Jafari, Ashwinkumar Ganesan, Chetan Sai Kumar Thalisetty, Varun Sivasubramanian, Tim Oates, and Tinoosh Mohsenin. “Sensornet: A Scalable and Low-Power Deep Convolutional Neural Network for Multimodal Data Classification”. *Transactions on Circuits and Systems I: Regular Papers* 66.1 (2018), pages 274–287. DOI: 10.1109/TCSI.2018.2848647.

[60] S Rallapalli, H Qiu, A Bency, S Karthikeyan, R Govindan, B Manjunath, and R Urgaonkar. “Are Very Deep Neural Networks Feasible on Mobile Devices”. *Trans. Circ. Syst. Video Technol* (2016).

[61] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. “An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads”. *International Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2017, pages 353–364. DOI: 10.1109/RTAS.2017.3.

[62] Travis Manderson, Juan Camilo Gamboa Higuera, Ran Cheng, and Gregory Dudek. “Vision-Based Autonomous Underwater Swimming in Dense Coral for Combined Collision Avoidance and Target Selection”. *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pages 1885–1891. DOI: 10.1109/IROS.2018.8594410.

[63] Shenshen Gu, Xinyi Chen, Wei Zeng, and Xin Wang. “A Deep Learning Tennis Ball Collection Robot and the Implementation on NVIDIA Jetson TX1 Board”. *International Conference on Advanced Intelligent Mechatronics (AIM)*. IEEE. 2018, pages 170–175. DOI: 10.1109/AIM.2018.8452263.

[64] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. "Middleware for Robotics: A Survey". *International Conference on Robotics, Automation and Mechatronics (ICMRA)*. IEEE. 2008, pages 736–742. DOI: 10.1109/RAMECH.2008.4681485.

[65] Herman Bruyninckx. "Open Robot Control Software: The OROCOS Project". *International Conference on Robotics and Automation (ICRA)*. IEEE. 2001, pages 2523–2528. DOI: 10.1109/ROBOT.2001.933002.

[66] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. "YARP: Yet Another Robot Platform". *International Advanced Robotic Systems* 3.1 (2006), page 8. DOI: 10.5772/5761.

[67] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the Performance of ROS2". *International Conference on Embedded Software (EMSOFT)*. 2016, pages 1–10. DOI: 10.1145/2968478.2968502.

[68] Kento Hasegawa, Kazunari Takasaki, Makoto Nishizawa, Ryota Ishikawa, Kazushi Kawamura, and Nozomu Togawa. "Implementation of a ROS-Based Autonomous Vehicle on an FPGA Board". *International Conference on Field Programmable Technology (FPT)*. 2019. DOI: 10.1109/ICFPT47387.2019.00092.

[69] Yasuhiro Nitta, Sou Tamura, Hidetoshi Yugen, and Hideki Takase. "ZytleBot: FPGA Integrated Development Platform for ROS Based Autonomous Mobile Robot". *International Conference on Field Programmable Technology (FPT)*. 2019. DOI: 10.1109/FPL.2019.00077.

[70] J. Peña Queralta, F. Yuhong, L. Salomaa, L. Qingqing, T. N. Gia, Z. Zou, H. Tenhunen, and T. Westerlund. "FPGA-Based Architecture for a Low-Cost 3D Lidar Design and Implementation from Multiple Rotating 2D Lidars with ROS". *Sensors*. IEEE. 2019, pages 1–4. DOI: 10.1109/SENSORS43011.2019.8956928.

[71] Stefano Aldegheri, Nicola Bombieri, Nicola Dall'Ora, Franco Fummi, Simone Girardi, and Marco Panato. "A Framework for the Design and Simulation of Embedded Vision Applications Based on OpenVX and ROS". *International Symposium on Circuits and Systems (ISCAS)*. 2018. DOI: 10.1109/ISCAS.2018.8351514.

[72] Kazushi Yamashina, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. "Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems". *CoRR* abs/1508.07123 (2015). arXiv: 1508.07123.

[73] Dayang NA Jawawi, Rosbi Mamat, and Safaai Deris. "A Component-Oriented Programming for Embedded Mobile Robot Software". *Advanced Robotic Systems* 4.3 (2007), page 40. DOI: 10.5772/5678.

[74] Kazushi Yamashina, Hitomi Kimura, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. "cReComp: Automated Design Tool for ROS-Compliant FPGA Component". *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE. 2016, pages 138–145. DOI: 10.1109/MCSoC.2016.47.

[75] Takeshi Ohkawa, Kazushi Yamashina, Takuya Matsumoto, Kanemitsu Ootsu, and Takashi Yokota. "Automatic Generation Tool of FPGA Components for Robots". *IEICE Transactions on Information and Systems* 102.5 (2019), pages 1012–1019. DOI: 10.1587/transinf.2018RCP0004.

[76] Takeshi Ohkawa, Kazushi Yamashina, Takuya Matsumoto, Kanemitsu Ootsu, and Takashi Yokota. "Architecture Exploration of Intelligent Robot System Using ROS-Compliant FPGA Component". *International Symposium on Rapid System Prototyping (RSP)*. IEEE. 2016, pages 1–7. DOI: 10.1145/2990299.2990312.

[77] Takeshi Ohkawa, Kazushi Yamashina, Hitomi Kimura, Kanemitsu Ootsu, and Takashi Yokota. “FPGA Components for Integrating FPGAs into Robot Systems”. *IEICE Transactions on Information and Systems* 101.2 (2018), pages 363–375. DOI: 10.1587/transinf.2017RCP0011.

[78] Yuhei Sugata, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. “Acceleration of Publish/Subscribe Messaging in ROS-Compliant FPGA Component”. *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. 2017, pages 1–6. DOI: 10.1145/3120895.3120904.

[79] David Sidler, Zsolt István, and Gustavo Alonso. “Low-latency TCP/IP stack for data center applications”. *International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pages 1–4. DOI: 10.1109/FPL.2016.7577319.

[80] Daniel Pinheiro Leal, Midori Sugaya, Hideharu Amano, and Takeshi Ohkawa. “FPGA Acceleration of ROS2-Based Reinforcement Learning Agents”. *International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. 2020, pages 106–112. DOI: 10.1109/CANDARW51189.2020.00031.

[81] Marc Eisoldt, Steffen Hinderink, Marco Tassemeier, Marcel Flottmann, Juri Vana, Thomas Wiemann, Julian Gaal, Marc Rothmann, and Mario Porrmann. “ReconfROS: Running ROS on Reconfigurable SoCs”. *Workshop on Drone Systems Engineering. Drone Systems Engineering (DroneSE)*. 2021, pages 16–21. DOI: 10.1145/3444950.3444959.

[82] Takeshi Ohkawa, Yuhei Sugata, Harumi Watanabe, Nobuhiko Ogura, Kanemitsu Ootsu, and Takashi Yokota. “High-Level Synthesis of ROS Protocol Interpretation and Communication Circuit for FPGA”. *International Workshop on Robotics Software Engineering (RoSE)*. IEEE. 2019, pages 33–36. DOI: 10.1109/RoSE.2019.00014.

[83] Hideki Takase, Tomoya Mori, Kazuyoshi Takagi, and Naofumi Takagi. “mROS: A Lightweight Runtime Environment for Robot Software Components onto Embedded Devices”. *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. 2019, pages 1–6. DOI: 10.1145/3337801.3337815.

[84] Hideki Takase, Tomoya Mori, Kazuyoshi Takagi, and Naofumi Takagi. “mROS: a Lightweight Runtime Environment of ROS 1 Nodes for Embedded Devices”. *Information Processing* 28 (2020), pages 150–160. DOI: 10.2197/ipsjjip.28.150.

[85] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. “Run-Time Power Gating in Hybrid ARM-FPGA Devices”. *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pages 1–6. DOI: 10.1109/FPL.2014.6927503.

[86] Christian Lienen, Marco Platzner, and Bernhard Rinner. “ReconROS: Flexible Hardware Acceleration for ROS2 Applications”. *International Conference on Field Programmable Technology (FPT)*. IEEE, 2020, pages 268–276. DOI: 10.1109/ICFPT51103.2020.00046.

[87] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. “ReconOS: An Operating System Approach for Reconfigurable Computing”. *Micro* 34.1 (2014), pages 60–71. DOI: 10.1109/MM.2013.110.

[88] Christian Lienen and Marco Platzner. “ReconROS Executor: Event-Driven Programming of FPGA-Accelerated ROS 2 Applications”. *CoRR* abs/2201.07454 (2022).

[89] Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. “A Survey of Model-Driven Engineering in Robotics”. *Computer Languages* 62 (2021), page 101021. DOI: 10.1016/j.cola.2020.101021.

[90] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley and Sons, Inc., 2006.

[91] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. "A Survey on Domain-Specific Languages in Robotics". *Simulation, Modeling, and Programming for Autonomous Robots*. Springer International Publishing, 2014, pages 195–206. DOI: 10.1007/978-3-319-11900-7_17.

[92] Ruediger Willenberg, Zamira Daw, Christian Englert, and Marcus Vetter. "Generation of Deterministic MCU/FPGA Hybrid Systems from UML Activities". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2010, pages 340–345. DOI: 10.1109/FPL.2010.74.

[93] Chiraz Trabelsi, Samy Meftali, and Jean-Luc Dekeyser. "Decentralized Control for Dynamically Reconfigurable FPGA Systems". *Microprocessors and Microsystems* 37.8 (2013), pages 871–884. DOI: 10.1016/j.micpro.2013.04.012.

[94] Chiraz Trabelsi, Samy Mettali, Rabie ben Atitallah, and Jean-Luc Dekeyser. "Model-Driven Design Flow for Distributed Control in Reconfigurable FPGA Systems". *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2014, pages 1–6. DOI: 10.1109/DASIP.2014.7115631.

[95] Chiraz Trabelsi, Samy Meftali, and Jean-Luc Dekeyser. "Distributed Control for Reconfigurable FPGA Systems: A High-Level Design Approach". *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE. 2012, pages 1–8. DOI: 10.1109/ReCoSoC.2012.6322871.

[96] Remigiusz Wiśniewski, Grzegorz Bazydło, Luis Gomes, and Aniko Costa. "Dynamic Partial Reconfiguration of Concurrent Control Systems Implemented in FPGA Devices". *IEEE Transactions on Industrial Informatics* 13.4 (2017), pages 1734–1741. DOI: 10.1109/TII.2017.2702564.

[97] Vladimir Estivill-Castro, René Hexel, and Morgan McColl. "High-Level Executable Models of Reactive Real-Time Systems with Logic-Labelled Finite-State Machines and FPGAs". *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2018, pages 1–8. DOI: 10.1109/RECONFIG.2018.8641710.

[98] Vladimir Estivill-Castro and René Hexel. "Arrangements of Finite-State Machines Semantics, Simulation, and Model Checking". *International Conference on Model-Driven Engineering and Software Development*. Volume 2. 2013, pages 182–189. DOI: 10.5220/0004317101820189.

[99] Taylor Riché, Jim Nagle, Joyce Xu, and Don Hubbard. "Converting Executable Floating-Point Models to Executable and Synthesizable Fixed-Point Models". *International Conference on Model-Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2019, pages 354–361. DOI: 10.1109/MODELS-C.2019.00055.

[100] Mouna Baklouti, Manel Ammar, Philippe Marquet, Mohamed Abid, and Jean-Luc Dekeyser. "A Model-Driven Based Framework for Rapid Parallel SoC FPGA Prototyping". *International Symposium on Rapid System Prototyping (RSP)*. IEEE. 2011, pages 149–155. DOI: 10.1109/RSP.2011.5929989.

[101] Ciprian Teodorov, Damien Picard, and Loic Lagadec. "FPGA Physical-Design Automation Using Model-Driven Engineering". *International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC)*. IEEE. 2011, pages 1–6. DOI: 10.1109/ReCoSoC.2011.5981495.

[102] Roberto de Medeiros, Marcilyanne M Gois, Drausio L Rossi, and Vanderlei Bonato. "Designing Embedded Systems with MARTE: A PIM to PSM Converter". *International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2012, pages 303–306. DOI: 10.1109/SIES.2012.6356602.

[103] Marcela Leite, Cristiano D Vasconcellos, and Marco Aurélio Wehrmeister. "Enhancing Automatic Generation of VHDL DescrIPtions from UML/MARTE Models". *International Conference on Industrial Informatics (INDIN)*. IEEE. 2014, pages 152–157. DOI: 10.1109/INDIN.2014.6945500.

[104] Marcela Leite and Marco Aurélio Wehrmeister. "Aspect-Oriented Model-Driven Engineering for FPGA/VHDL Based Embedded Real-Time Systems". *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE. 2014, pages 261–268. DOI: 10.1109/ISORC.2014.45.

[105] Huafeng Zhang, Yu Jiang, Han Liu, Hehua Zhang, Ming Gu, and Jiaguang Sun. "Model-Driven Design of Heterogeneous Synchronous Embedded Systems". *International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pages 774–779. DOI: 10.1145/2970276.2970280.

[106] Franz-Josef Streit, Martin Letras, Stefan Wildermann, Benjamin Hackenberg, Joachim Falk, Andreas Becher, and Jürgen Teich. "Model-Based Design Automation of Hardware/software Co-Designs for Xilinx Zynq MPSoC". *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2018, pages 1–8. DOI: 10.1109/RECONFIG.2018.8641736.

[107] Andrea Enrici, Julien Lallet, Renaud Pacalet, Ludovic Apvrille, Karol Desnos, and Imran Latif. "Model-Based Programming for Multi-Processor Platforms with TTool/DIPLODOCUS and OMC". *International Conference on Model-Driven Engineering and Software Development*. Springer. 2018, pages 56–81. DOI: 10.1007/978-3-030-11030-7_4.

[108] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Jean-Philippe Diguet, and Sebastien Guillet. "Dynamic Applications on Reconfigurable Systems: from UML Model Design to FPGAs Implementation". *International Conference on Design, Automation and Test in Europe (DATE)*. IEEE. 2011, pages 1–4. DOI: 10.1109/DATE.2011.5763315.

[109] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Jean-Philippe Diguet, and Philippe Soulard. "UML Design for Dynamically Reconfigurable MultIProcessor Embedded Systems". *International Conference on Design, Automation and Test in Europe (DATE)*. IEEE. 2010, pages 1195–1200. DOI: 10.1109/DATE.2010.5456989.

[110] Gilberto Ochoa, El-Bay Bourennane, Hassan Rabah, and Ouassila Labbani. "High-Level Modelling and Automatic Generation of Dynamicaly Reconfigurable Systems". *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2011, pages 1–8. DOI: 10.1109/DASIP.2011.6136900.

[111] Gilberto Ochoa-Ruiz, Sébastien Guillet, Florent De Lamotte, Eric Rutten, El-Bay Bourennane, Jean-Philippe Diguet, and Guy Gogniat. "An UML Approach for Rapid Prototyping and Implementation of Dynamic Reconfigurable Systems". *Transactions on Design Automation of Electronic Systems (TODAES)* 21.1 (2015), pages 1–25. DOI: 10.1145/2800784.

[112] Youenn Corre, Jean-Philippe Diguet, Loic Lagadec, Dominique Heller, and Dominique Blouin. "Fast Template-Based Heterogeneous MPSoC Synthesis on FPGA". *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer. 2013, pages 154–166. DOI: 10.1007/978-3-642-36812-7_15.

[113] Wolfgang Ecker, Keerthikumara Devarajegowda, Michael Werner, Zhao Han, and Lorenzo Servadei. "Embedded Systems' Automation Following Omg's Model-Driven Architecture Vision". *International Conference on Design, Automation and Test in Europe (DATE)*. IEEE. 2019, pages 1301–1306. DOI: 10.23919/DATE.2019.8715154.

[114] Johannes Wienke, Arne Nordmann, and Sebastian Wrede. "A Meta-model and Toolchain for Improved Interoperability of Robotic Frameworks". *Simulation, Modeling, and Programming for Autonomous Robots*. Springer Berlin Heidelberg, 2012, pages 323–334. DOI: 10.1007/978-3-642-34327-8_30.

[115] Fábio M Costa, Karl A Morris, Fabio Kon, and Peter J Clarke. "Model-Driven Domain-Specific Middleware". *International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pages 1961–1971. DOI: 10.1109/ICDCS.2017.197.

[116] Donald E Knuth. "Semantics of Context-Free Languages". *Mathematical systems theory* 2.2 (1968), pages 127–145. DOI: 10.1007/BF01692511.

[117] R. Farrow. "Generating a Production Compiler from an Attribute Grammar". English. *Software* 1.04 (Oct. 1984), pages 77–93. DOI: 10.1109/MS.1984.229467.

[118] Christoff Bürger, Sven Karol, and Christian Wende. "Applying Attribute Grammars for Metamodel Semantics". *International Workshop on Formalization of Modeling Languages*. ACM, 2010, page 1. DOI: 10.1145/1943397.1943398.

[119] Jesper Öqvist. "ExtendJ: Extensible Java compiler". *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Programming'18 Companion. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pages 234–235. DOI: 10.1145/3191697.3213798.

[120] Harald H Vogt, S Doaitse Swierstra, and Matthijs F Kuiper. "Higher Order Attribute Grammars". *SIGPLAN Notices* 24.7 (1989), pages 131–145. DOI: 10.1145/73141.74830.

[121] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. "Relational Reference Attribute Grammars: Improving Continuous Model Validation". *Computer Languages* 57 (2020), page 100940. DOI: 10.1016/j.cola.2019.100940.

[122] Object Management Group (OMG). *Interface Definition Language, Version 4.2*. OMG Document Number formal/18-01-05 (https://www.omg.org/spec/IDL/4.2). 2018.

[123] Johannes Mey, Thomas Kühn, René Schöne, and Uwe Assmann. "Reusing Static Analysis across Different Domain-Specific Languages Using Reference Attribute Grammars". *The Art, Science, and Engineering of Programming* 4.3 (Feb. 1, 2020), 15:1–36. DOI: 10.22152/programming-journal.org/2020/4/15.

[124] Chris Wanstrath. *mustache - Logic-Less Templates*. https://mustache.github.io. Accessed: 2020-07-20. 2020.

[125] Jishnu Saurav Mittapalli and Menaka Pushpa Arthur. "Survey on Template Engines in Java". en. *ITM Web of Conferences* 37 (2021). Publisher: EDP Sciences, page 01007. DOI: 10.1051/itmconf/20213701007.

[126] Eve Coste-Maniere and Reid Simmons. "Architecture, the Backbone of Robotic Systems". *International Conference on Robotics and Automation (ICRA)*. Volume 1. IEEE. 2000, pages 67–72. DOI: 10.1109/ROBOT.2000.844041.

[127] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H Jones. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". *International Conference on Embedded Software and Systems (ICESS)*. IEEE. 2019, pages 1–8. DOI: 10.1109/ICESS.2019.8782524.

[128] Onur Ulusel, Christopher Picardo, Christopher B Harris, Sherief Reda, and R Iris Bahar. "Hardware Acceleration of Feature Detection and Description Algorithms on Low-Power Embedded Platforms". *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pages 1–9. DOI: 10.1109/FPL.2016.7577310.

[129] Stylianos I Venieris and Christos-Savvas Bouganis. "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs". *Transactions on Neural Networks and Learning Systems* 30.2 (2018), pages 326–342. DOI: 10.1109/TNNLS.2018.2844093.

[130] Stephanie Soldavini and Christian Pilato. "A Survey on Domain-Specific Memory Architectures". *arXiv preprint arXiv:2108.08672* (2021). DOI: 10.29292/jics.v16i2.509.

[131] D.C. Schmidt. "Model-Driven Engineering". *Journal Computer* 39.2 (2006), pages 25–31. DOI: 10.1109/MC.2006.58.

[132] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. "Declarative Intraprocedural Flow Analysis of Java Source Code". *Electronic Notes in Theoretical Computer Science* 238.5 (2009). International Workshop on Language Descriptions, Tools and Applications (LDTA), pages 155–171. DOI: https://doi.org/10.1016/j.entcs.2009.09.046.

[133] Luca Benini and Giovanni De Micheli. "Networks on Chip: A New Paradigm for Systems on Chip Design". *International Conference on Design, Automation and Test in Europe (DATE)*. IEEE. 2002, pages 418–419. DOI: 10.1109/DATE.2002.998307.

[134] Tobias Bjerregaard and Shankar Mahadevan. "A Survey of Research and Practices of Network-On-Chip". *Computing Surveys (CSUR)* 38.1 (2006), pages 1–52. DOI: 10.1145/1132952.1132953.

[135] Salma Hesham, Jens Rettkowski, Diana Goehringer, and Mohamed A Abd El Ghany. "Survey on Real-Time Networks-On-Chip". *Transactions on Parallel and Distributed Systems* 28.5 (2016), pages 1500–1517. DOI: 10.1109/TPDS.2016.2623619.

[136] Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. "Kilo-NoC: A Heterogeneous Network-On-Chip Architecture for Scalability and Service Guarantees". *International Symposium on on Computer Architecture (ISCA)*. IEEE. 2011, pages 401–412. DOI: 10.1145/2000064.2000112.

[137] Habib Khan, Ariel Podlubne, and Diana Göhringer. "Intrusive FPGA-in-the-loop debugging using a rule-based inference system". *Microprocessors and Microsystems* 64 (2019), pages 185–194. DOI: 10.1016/j.micpro.2018.11.004.

[138] Habib Khan, Gökhan Akgün, Ariel Podlubne, Felix Wegener, Amir Moradi, Diana Göhringer, et al. "Cycle-Accurate Debugging of Multi-clock Reconfigurable Systems". *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2019, pages 1–5. DOI: 10.1109/ReConFig48160.2019.8994806.

[139] Habib Khan, Ariel Podlubne, Gökhan Akgün, Diana Göhringer, et al. "Cycle-Accurate Debugging of Embedded Designs Using Recurrent Neural Networks". *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer. 2020, pages 1–5. DOI: 10.1007/978-3-030-44534-8_6.

[140] Ronny Seiger et al. “Immersives verteiltes Robotic Co-working”. *Informatik Spektrum* 43.6 (2020), pages 425–435. DOI: 10.1007/s00287-020-01297-w.

[141] Ahmad Sadek, Ananya Muddukrishna, Lester Kalms, Asbjørn Djupdal, Ariel Podlubne, Antonio Paolillo, Diana Goehringer, and Magnus Jahre. “Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview”. *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer. 2018, pages 737–749. DOI: 10.1007/978-3-319-78890-6_59.

[142] Ariel Podlubne, Julian Haase, Lester Kalms, Gökhan Akgün, Muhammad Ali, Habib Ulhasan Khar, Ahmed Kamal, and Diana Göhringer. “Low Power Image Processing Applications on FPGAs Using Dynamic Voltage Scaling and Partial Reconfiguration”. *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2018, pages 64–69. DOI: 10.1109/DASIP.2018.8596910.

[143] Safdar Mahmood et al. “Prospects of Robots in Assisted Living Environment”. *Electronics* 10.17 (2021), page 2062. DOI: 10.3390/electronics10172062.

[144] Tina Bobbe, Hans Winger, Ariel Podlubne, Florian Wieczorek, Lisa-Marie Lüneburg, Ievgen Kharabet, Jens Wagner, and Sergio Pertuz. “Reflections on "Rock, Paper, Scissors": Communicating Science to the Public through a Demonstrator”. *International Conference on Human-Robot Interaction (HRI)*. IEEE, 2022, pages 1208–1209. DOI: 10.1109/HRI53351.2022.9889613.

[145] Johannes Mey, René Schöne, Ariel Podlubne, and Uwe Aßmann. “Specifying Reactive Robotic Applications With Reference Attribute Motion Grammars”. *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2022. DOI: 10.1109/ACSOSC56246.2022.00035.

[146] Sergio Pertuz, Ariel Podlubne, and Diana Göhringer. “An Efficient Accelerator for Nonlinear Model Predictive Control”. *International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*. IEEE. 2023, pages 1–8. DOI: 10.1109/ASAP57973.2023.00038.