Model-based Approach for Automatic Generation of Hardware Architectures for Robotics

ARIEL PODLUBNE¹,³, JOHANNES MEY²,³, RENÉ SCHÖNE², UWE AßMANN²,³ and DIANA GÖHRINGER¹,³

¹Chair of Adaptive Dynamic Systems
²Chair of Software Technology
³Centre for Tactile Internet with Human-in-the-Loop (CeTI) - Technische Universität Dresden (TUD)
{ariel.podlubne, johannes.mey, rene.schoene, uwe.assmann, diana.goehringer}@tu-dresden.de

Corresponding author: Ariel Podlubne (e-mail: ariel.podlubne@tu-dresden.de).

This work has been 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.

ABSTRACT Increasingly complex robotic platforms incorporate heterogeneous sensors and actuators. They are usually coupled with embedded computers but relying on software solutions not entirely suited for processing large amount of data concurrently and fast enough to keep real-time constrains. FPGAs are ideal candidates to enhance those systems computing capabilities while still being programmable. However, it is cumbersome to manually incorporate them into new or existing systems because providing accelerators with a specific integration capability limits their applicability. This work presents an approach to integrate multiple hardware accelerators into a robotic system. A model-based toolchain automatically generates the necessary hardware components (VHDL modules) from existing message specifications to exchange data with the accelerators. The goal is to seamlessly replace software components with FPGA-based ones while retaining the same communication interface. Instead of writing several hundred lines of VHDL, a dozen input specification lines are sufficient with our approach. The results are validated through an evaluation of all message specifications included in the latest Robot Operating System (ROS) versions. The 2295 messages evaluated show the robustness in our capabilities to support arbitrarily large ROS messages types, multiple data types, and nested messages. Moreover, our approach facilitates the extension from ROS1 to provide support for ROS2 easily. Finally, two use cases are shown to prove the feasibility of our approach on real applications. The first one incorporates a High Level Synthesis (HLS) hardware accelerator for image processing into an existing software architecture. The second one consists of a fully FPGA-based mobile platform with ROS features incorporated.

INDEX TERMS Robotics, Embedded Hardware, FPGA, ROS, ROS2, Model-Based, Code Generation, Attribute Grammar.

I. INTRODUCTION Heterogeneous computing has grown over the last years, improving the innovations on accelerating compute-intensive workloads such as artificial intelligence [1]. The field of computer architectures has become quite diverse with the emergence and constant improvements of Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and Field Programmable Gate Arrays (FPGAs). FPGAs 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 combine them with software solutions. Therefore, designing scalable and reusable interfaces between these two is desired to achieve good synergy between FPGAs and software systems. FPGAs offer higher performance and higher energy-efficiency compared to CPUs and GPUs for tasks such as feature detection and description algorithms [2] or convolutional neural networks [3]. FPGAs can be, from the per-
formance per watt, close to Application-Specific Integrated Circuits (ASICs) [4], [5] but they are more flexible as they are reprogrammable.

The embedded systems community has also focused its attention towards FPGAs [6], bringing new tools and programming paradigms, primarily based on High Level Synthesis (HLS). Different commercial (e.g., Xilinx OpenCV, Matlab HDL Coder), as well as academic frameworks [7], [8] are available, providing multiple functions as individual elements (e.g., filters) to be integrated into a given architecture.

Nowadays, multiple heterogeneous robotic platforms such as self-driving cars or drones [9], [10] use lidars, radars, various cameras, and other types of sensors (e.g., Inertial Measurement Units (IMUs), ultrasonic) in combination to sense their surroundings. This implies that a large amount of heterogeneous data is continuously generated, and it is likely required to be processed in real-time. FPGAs are an ideal candidate due to their parallel architecture. They provide versatility to design hardware according to the needs precisely and can pre-process data very close to sensors [11], making them good candidates for robotic applications.

The robotics community has adopted Robot Operating System (ROS) as the mainstream middleware over the last years.1 There are standard algorithms for basic tasks such as localization, control, or mapping 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 these allows to design more complex robotic systems, demanding more computational power and often needing to process large amount of data in parallel. However, despite all these and the advantages of FPGAs mentioned, 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. This would ease their utilization by other fields such as robotics and improve their processing capabilities. However, these impose the following challenges:

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.

2) Complex Architectures: Manage multiple accelerators and their communication with other components from the distributed system.

3) Adaptivity: Provide flexibility to be extensible for new features, hardware components (e.g., sensors and actuators), and middlewares.

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 contributions of this work are a flexible model-based workflow to generate hardware architectures for robotics based on 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 rest of the paper is organized as follows. Section II explores the related work. The Reconfigurable Computing System (RCS) architecture for robotic applications is described in Section III. Section IV introduces a model-based approach aiming to solve the challenges mentioned in this section. Section V evaluates the toolchain’s capabilities and shows its advantages with two different ROS-based robotic uses cases. The paper is concluded in Section VI.

II. RELATED WORK

Porting robotic applications originally designed in software to hardware accelerators requires suitable interfaces which depend on given specifications. This section describes related work on approaches to integrate specific robotic applications to FPGAs, frameworks, and modeling techniques to fulfill those specifications.

The main components of a ROS-based robotics system (called nodes) are characterized by the messages they can send and receive. They include multiple off-the-shelf types of specifications, closely related to sensors and actuators’ needs (e.g., IMUs, images, velocities), and also support custom ones. The flexibility of nodes and messages allows for the reusability of ROS components to easily deploy algorithms in a distributed software system.

ROS is classically designed to run on a CPU/GPU system. However, several works have been proposed to combine FPGAs with it [12], [13]. They mainly rely on Xilinx’s System-on-Chip (SoC) FPGAs which are capable of running Linux on their Processing System (PS). Therefore, these designs cannot be realized in the absence of a PS capable of running an Operating System (OS) as Linux.

Some related works mainly focus on specific applications, accelerating some parts of a ROS-based software implementation. Queralta et al. [14] proposed a low cost 3D Lidar-based design. The authors rely on low-cost sensors to obtain 3D point clouds for localization and mapping algorithms. All the processing is implemented in Very High Speed

---

1 ROS Community Metrics available at http://wiki.ros.org/Metrics
Integrated Circuit Hardware Description Language (VHDL). However, the communication with ROS, running on a PC, is done via the Universal Asynchronous Receiver-Transmitter (UART) interface which could introduce some overhead. Also, it is a slow communication protocol compared to the processing speed achieved by FPGAs which diminishes its advantages.

Ohkawa et al. [17] build upon previous work [16] 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 FPGA. Their work can be considered a generic approach as it takes ROS definitions targeting reusability, despite being limited to HLS implementations.

Leal et al. [18] provide a tool that relies on the open-source PYNQ project 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. So, it eases the integration of accelerators but not directly to its software counterpart as a bridge or interface to other ROS messages present in the system would still be needed. Even though they target ROS2, they use .msg and nothing precisely from ROS2 (such as the subset of OMG IDL specification 2). However, both ROS versions support .msg (with minor changes), and their approach is valid. They claim to support multiple data types. However, messages could be formed by variables of different data types or multiple nested messages which they seem to ignore as there is only support for (arrays of) primitive types.

Eisoldt et al. [19] focus on the integration of accelerators from the algorithmic point of view. Similar to previous references, there is a dependency of an embedded processor compatible of running an OS such as Linux. Hence, 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 over dedicated memory ports (not specified but assumed to be AXI Stream (AXIS)). There are references to specific HLS-related registers (e.g., AP_DONE, AP_CTRL) but no reference of Hardware Description Language (HDL) accelerators is mentioned.

Lienen et al. [20] highlight the lack of a consistent programming model for implementing software and hardware functions. They close that gap by integrating ROS to their previous work ReconOS [21], a tailored OS for multi-threaded programming of hardware and software threads for reconfigurable computers. Similarly to [19], they also relied on the Linux virtual address space and shared memory to exchange data between PS and PL.

We previously proposed in [24] a generic architecture to have a full hardware implementation without the need of any CPU. It is generic so that it can incorporate IPs designed in HDL or HLS. Moreover, it leaves the possibility to replace the communication block if a different device is used, or communications are handled by a CPU if available or desired. In case a different middleware rather than ROS is needed, the modular architecture allows to replace 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 [19]. However, there was not a systematic approach to map complete ROS messages to hardware, which is addressed in this work.

The related work presented so far demonstrated that accelerating ROS components using FPGAs is possible and beneficial. However, different robotics middleware besides ROS have emerged over the years, such as YARP [25] or OROCOS [26]. Having many options to choose from could become a disadvantage due to incompatibilities that may arise because components usually only can be reused easily within the middleware they have been developed for. The use of model-driven engineering rises the level of

2IDL - Interface Definition and Language Mapping available at https://design.ros2.org/articles/idl_interface_definition.html
abstraction, potentially solving this issue, by 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). Additionally, model-driven engineering speeds up the development process and the formalization of such abstractions enables the use of automated tools to verify the consistency of the generated artifacts, improving the reliability.

Costa et al. [22] proposed the use of model-driven engineering concepts to develop specialized middlewares for particular application domains. The approach follows a building-blocks concept. Several Meta-models are combined to create models to specify the configuration of the targeted middleware. The authors showed the feasibility of their approach with four domain-specific applications. However, an analysis of the complexity of data structures is not detailed.

Due to the fact that multiple robotics middlewares are available, Wienke et al. [23] proposed the use of model-based techniques for component reusability. They addressed data type compatibility in a structured way through the development of a generic meta-model capable of representing data types from different middlewares and their relations. This is possible thanks to the fact that 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. The authors performed a thorough analysis of the available representation features of common Interface Definition Languages (IDLs) based serialization mechanisms (statically typed) and the three robotics middlewares mentioned before (dynamically typed systems). A similar approach was followed in Section V.

Even though [18], [20], [24] show actual use cases, and [23] performed a detailed analysis of IDLs and several middlewares, no results for complex message types or even nested messages have been shown. To the best of our knowledge, we are the first to fully support them, for several middlewares, as shown in Section V-B.

Table 1 summarizes the approaches mentioned above, including those that combine FPGAs and ROS to improve the computational power of robotic systems as well as the non-FPGA ones, focusing on middlewares. It also highlights the need to extend the validation methodology beyond use cases. Note that n/a means the characteristic does not apply to that work.

### III. RECONFIGURABLE COMPUTING SYSTEMS FOR ROBOTIC APPLICATIONS

Components within a heterogeneous distributed system, such as a robotic platform, usually exchange data. They can either generate or consume it, and 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 1. The main requirement is that external components do not make any distinctions with hardware accelerators, so they can be interchangeable if needed.

Most elements in the proposed hardware architecture are foreseen to be on the PL side but not limited as some functionalities can reside on the PS side or external peripherals (e.g., communication from/to outside the FPGA [24]). The blocks shown in Figure 1 can be classified as: (1) **accelerator-related components** comprising the base architecture needed to handle multiple hardware accelerators (Challenge 2: Complex Architectures). Their design

### TABLE 1: Comparison against related work.

| Characteristics                  | [12], [13] | [14] | [15] | [17] | [18] | [19] | [20] | [22] | [23] | Previous Work [24] | Proposed Work |
|----------------------------------|------------|------|------|------|------|------|------|------|------|-------------------|---------------|
| Generalized approach             | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| CPU independent                   | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| Handle multiple hardware accelerators | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| Integration in existing ROS systems | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| Adaptability to other systems     | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| Supports multiple middlewares     | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |
| Supports logic validation         | ✓          | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓    | ✓                 | ✓             |

FIGURE 1: Proposed hardware architecture

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
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 (Challenge 1: **Interface Compliance**).

**Accelerator-related Components**

They are responsible for exchanging data between accelerators in FPGAs and external parts of the distributed system. The AXIS protocol is chosen to be the standard interface among all blocks due to its simplicity and widespread usage. The main component is the Manager block (Figure 2). Internally, it includes an AXIS demultiplexer (Com to IPs) and a multiplexer (IPs to Com) because there could be N publishers or subscribers (as hardware accelerator components). However, there is only one bi-directional channel to exchange data with the Communication Interface. Therefore, an Arbiter is required to handle these N accelerators. The Least Recently Used cache model is used as arbitration scheme to ensure that each accelerator can stream data at least once per round to avoid resource starvation. The entity port (input and out 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 there are multiple modules involved, raising the need to automate their generation (Challenge 3: **Adaptivity**).

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 to port to other vendors or platforms (e.g., 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 Lookup Tables (LUTs) and 300 Flip-Flops (FFs) for the simplest configuration. A similar solution following our approach uses 364 LUTs and 67 FFs but for 4 times the number of slaves, allowing to manage more accelerators with roughly the same resource utilization.

The block handling the communication (Communication Interface) is inspired by the TCP/IP Five-Layer Network Model, 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 standard) 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, the use of the PS is not mandatory but optional. We showed the feasibility of this in [24] where ROS was not running under Linux. The Communication Interface was in the PL as a Serial Peripheral Interface (SPI)-controlled device handling communications over Ethernet with external components. We further show the versatility of the architecture in Section V-C with one of the use cases having ROS running on the PS.

**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. 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 shows an example of an accelerator taking the role of a publisher with its interface component for the sensor_msgs/Image message specification (Listing 7). 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 certain characteristics of the techniques showed, 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. startIndx in Figure 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 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.
differ between ROS 1 and ROS 2 (in which furthermore different middlewares and thus serializations can be used), but also VHDL might require different revisions when using different synthesizers or hardware from different vendors.

Therefore, we propose a flexible and extensible model-driven toolchain to generate hardware interfaces for ROS components.

A Model-Driven Toolchain

Model-driven engineering [27] describes the technique to use a staged model transformation process in which models are transformed in iterations. Each additional information added to the final model is used to generate the desired code artifacts, which in our case is a set of VHDL components. The remainder of this section explains the proposed process and highlights the benefits of model-driven engineering and model-based code generation.

Figure 4 gives an overview of the code generation workflow, centering around the model-driven process implemented in our FPGA Interfaces for Robotic Middlewares (FIRM) tool.4

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

The main input is a single configuration file. An example of such a file is shown in Listing 1. Besides some configuration options for the hardware platform (Lines 1 to 6), 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 7 to 13). This configuration is parsed into a configuration model, which is used to generate both the accelerator-related 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 1 or ROS 2 is used, this is not resembled in the configuration file since it can be deduced from the context in which the

4FIRM is open source and it will be available subject to acceptance

Listing 1: Configuration for image processing use case

```
project:
name: Sobel Filter
platform:
board: zcu104
FPGA: xc7u2ev-ffgc1156-2-e
family: UltraScale
messages:
- type: ROS  # supported middleware
  name: sensor_msgs/Image
subscribers:
- AXIS2Image  # names for interfaces
publishers:
- Image2AXIS  # to middleware used by accelerators
```

3http://wiki.ros.org/gencpp
tool is run, which adds portability. Because the generated components require information about the structure of the ROS messages which serve as interfaces to the accelerator, 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 5a). Note that Figure 4 shows the possibility to extend FIRM with other middlewares besides ROS 1 and ROS 2 (Challenge 3: Adaptivity) following the same approach described here.

Using these models as well as the configuration model, analysis has to be performed in preparation of the interface generation (i.e., to compute the relative positions of its elements), required in the message-dependent IP cores. Because this analysis does not have to be aware of all details contained in the ROS msg format and should ideally be reusable for different message specifications, the ROS msg 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 5b). Using only this intermediate model and the configuration, the code for the accelerator-related components can be generated, decoupling the code generation from the message specification.

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. It is configured by files produced by generator components 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 template configurations are generated by our toolchain, the templates must be defined manually once, but 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 as well as their relation to the challenges presented in Section I are described next.

**Characteristics of the Model-Driven Toolchain**

The toolchain is modeled using Reference Attribute Grammars (RAG) [28] and implemented using the RAG system JastAdd [29]. Grammars specify a language using tokens (also known as terminal symbols) and non-terminal types with production rules that define 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 Abstract Syntax Tree (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 [30] – 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 construction of compilers [31]–[33], which our approach classifies for as well, because we transform a source language (a configuration file and a message specification) into a target language (VHDL). In our toolchain, the general idea for the use of attributes is the generation of 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 [34], computing entire derived models rather than simple properties.

To be able to handle (graph-shaped rather than tree-shaped) conceptual models such as the configuration and message models described previously more conveniently, we employ a relational RAG extension [35] adding a secondary graph structure to the AST, thus allowing more concise attribute specifications when dealing with nested message types and field types.

Using RAGs, our approach addresses Challenge 2: Complex Architectures and Challenge 3: Adaptivity. 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, we present a closer look at the employed models and attributes to further illustrate the workings and benefits of the chosen approach.

The Models
Figure 5 shows a meta-model representation of the employed models. While a grammar-based technique is used for clarity we chose to show the models in a graphical UML notation where grammar productions 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 5a 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 6.

In fact, there are two meta-models, one for each ROS version (cf. Section V-B) – the ROS 2 metamodel extends the ROS 1 metamodel with additional types. The aspect-oriented specification of attribute grammars [29] is used to extend the grammar. Additional elements are simply added using a ROS 2 grammar module; additional attributes and attribute equations are added using a grammar aspect.
While the ROS 2 communication system got a complete overhaul (internally, its messages are defined in the Interface Definition Language [36]), the (concrete) syntax is mostly backwards compatible, so a common parser can be used. Likewise, the model (or abstract syntax) for ROS 1 just needs minor extension to also support the features of ROS 2 as shown in Figure 5a. The three changes (default values, bounded strings, and arrays) are highlighted. The complete support for both versions of ROS addresses Challenge 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 

\[
\text{VhdlModel one Message, which itself contains a list of fields, which may contain atomic data types or (variable-length) streams of messages or data types. All other structural features are mapped to these features; fixed-length arrays, for example, are simply unrolled to a sequence of fields. Thus, this model does not depend on details of the ROS message format, creating an extension point for other message specifications, as proposed in [37].}
\]

This contributes to the intermediate model for the sensor_msgs/image message is shown in Figure 7.

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 has also 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 previously mentioned – the employed engine mustache [38] is logic-less (templates do not contain computations). Thus, all computations are done before or during the construction of the template configuration files within the presented tool, using its declarative static analysis capabilities. This not only results in 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 3: Adaptivity. We employ mustache as template engine, since it is a mature tool fitting our requirements [39].

Attributes
So far, the structure of the models was presented. Next, it is demonstrated how attributes are used to perform the model transformation and the analysis required for it. The computation of the attributes bitwidth and StartIndex (as shown in Figure 3) will serve as examples for the analysis. Listing 2 shows the attribute bitwidth, computing the bitwidth of the individual fields in the AXIS, required during code generation. It is a synthesized attribute [30], 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 Section III is StartIndex shown in Listing 3, which 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. For the following fields in the sub-stream, their index starts again at 0. This is computed by an inherited attribute obtaining information from its context, i.e., its parents in the abstract syntax tree [30]. In this case, the required context is the

![FIGURE 6: ROS message model for sensor_msgs/Image](image)
Listing 3: Declaration and definition of the inherited attribute \( \text{startIndex} \)

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

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.

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 [34] that compute entire (sub-)trees. In the JastAdd tool, these attributes are also called nonterminal attributes (NTA). Figure 8 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 using the information and available attributes from the nonterminal they are defined on. Finally, a (synthesized) attribute \text{print} is used to obtain a string representation of the template configuration which can be used by the template engine. Listing 4 shows (parts of) the result of this code generation for the image message (Figure 3), including values computed using the presented attributes.

Template-based Code Generation

The configuration shown in Listing 4 – a YAML file – configures the mustache template engine, which finally expands a set of templates we call 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 and Listing 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.

V. EVALUATION

The evaluation of the workflow presented in previous sections has been split in three parts. Section V-A analyzes the complexity of message specifications support by FIRM. Section V-B shows the evaluation performed based on real message specifications included in both ROS versions supported and highlights the possibility of our approach to also perform logic validation of the autogenerated components. Lastly, Section V-C shows two use cases with different accelerators, as publishers and subscribers, being part of a ROS system.
A. COMPLEXITY OF SPECIFICATIONS

The first step to assure 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 [23], a study of the datatypes supported by ROS 1 and ROS 2 is performed and then compared to the features supported by FIRM, presented in previous sections. The main characteristics with respect to datatypes are shown in Table 2. It depicts that our approach supports all characteristics of ROS 1 and ROS 2 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 in Section V-B.

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 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 2, as they could become a new datatype considering that only their bitwidth needs to be specified.

Even though Unions are not supported by ROS 1 nor ROS 2, they could be potentially added to our workflow by transmitting its width along with the data and 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 a 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 9. All their possible combinations is what leads 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 Figure 3. Then follow the messages in level ExpandingMessages. These can be built-in types declared as fixed sizes array, which are unfolded; inline elements of SimpleMessage or a combination of these two as bounded sized arrays of messages which are unfolded and inline. An example of the inlining is header in Figure 3, 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. Note that strings also form

FIGURE 8: Model transformation and code generation attributes

FIGURE 9: Complexity of ROS messages
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 Figure 3. 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 special 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 the generation of all sorts of complex message specifications with the proposed workflow. Specific design rules, shown in Table 2 and explained previously, were followed based on this to cope with the complexity and support all the combinations of messages shown in Figure 9 by FIRM.

### TABLE 2: Supported datatypes, ✓ mark supported, ✗ unsupported, and ⊕ potentially addable features.

|   | ROS 1 | ROS 2 | 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 | ✗ ✗ ⊕ | ✗ ✗ ⊕ |
| 6 | Constants | ✓ ✓ ✓ (not transmitted) | ✓ ✓ ✓ (not transmitted) |
| 7 | Enums | ✓ ✓ ✓ | ✓ ✓ ✓ |
| 8 | Variable Length Arrays | ✓ ✓ ✓ | ✓ ✓ ✓ (value always transmitted) |
| 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 [23].

### B. FULL ROS SUPPORT

Individual experiments were performed on all messages included in the latest three ROS 1 distributions (Kinetic, Melodic, and Noetic) and ROS 2’s LTS Foxy, in order to prove our support for the characteristics highlighted in Table 2. Considering the diversity of these messages, three groups, namely amount of elements, amount of different data types and depth of nested messages were considered. These three characteristics encompass all levels of complexity shown in Figure 9. Experiments showed that our generic model-based approach supports arbitrarily large ROS messages, multiple data types and nested messages. Figure 10 shows the histograms for these three characteristics, based on the evaluated ROS messages used in these experiments. Note that ROS 1 and ROS 2 provide a base set of packages which include about 200 message specifications. Experiments for them in all ROS versions mentioned before were performed, obtaining similar results for all of them. Additionally, to evaluate the robustness of our approach, extra messages from the ROS open source community were used. All installable packages of Noetic (latest ROS 1 distribution) add up to 2295 distinct message specifications, which are shown in Figures 10 and 11.

Each experiment consisted of an autogenerated project for each ROS message to run post-synthesis simulation to validate the logic of the interfaces. Therein, each project included: (1) the hardware interfaces (both directions) for a specific ROS message, considered the Device Under Tests (DUTs), (2) the remaining components of the base architecture shown in Figure 1, and (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 accordingly for the latter one to read and generate an out AXIS frame. A successful test implies that both AXIS streams are equal. In a real application, the accelerators that perform computation would be in between these two hardware interfaces in a publisher/subscriber combination or as shown in Figure 1.

Both DUTs are generated with the combination of template configurations generated from the intermediate repro-
sentation in *FIRM* and the HDTs, as shown in Figure 4. The *Accelerator-related Components* (Section III) are the same for every project of each ROS message. This is designed like that to have a similar scenario as an actual application with the *Accelerator-related Components* (similarly to the use cases shown in Section V-C) instead of just using the DUTs.

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 4), 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 have already available the intermediate representation of the ROS message to be evaluated. However, as specific fields in the message may have variable length (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 (Figure 10(a)) ensures multiple message lengths. (2) for cases in which unconstrained elements (Figure 11) or nested messages are included (Figure 10(c)). These two points ensure the extensive evaluation of arbitrarily large ROS messages. Figure 10(b) shows a comprehensive coverage of tested messages composed by *multiple distinct variable types*, ensuring our 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 (Figure 10a), including mainly 20 distinct types (Figure 10b) or most commonly with 9 levels of nested messages (Figure 10c). However, they show that our generic approach is not limited to even larger or more complex messages with more distinct fields or multiple levels of nested messages. It is only that they are not that common or included in the installable packages used for these experiments.

A systematic approach was followed during the design process to *manually* generate a set of message specifications to cover all of possibly infinite number of them. This left out cases that were not considered which is why our current 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 2. 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 stimulus but this can
be extended to generate more stimuli for each message, randomly or systematically to extend coverage. This implies the generation of multiple stimuli for each message, which would increase the total time to run all the experiments (currently over 41 hours with one stimulus for each 2295 messages).

Extension for ROS2 support
So far, the evaluation has been shown with ROS 1 Noetic distribution as mentioned previously. However, the possibility to easily extend the support for ROS 2 is simplified due to the nature of our approach with a model-based design of FIRM that generates the intermediate message model plus the ability to design tailored templates. The use of .msg for message specifications brings backward compatibility. Nevertheless, the communication scheme in ROS 2 is the main characteristic of this new version and the serialization of ROS messages differs slightly. This leads to a new set of HDTs adapted to take it into account for ROS 2 support.

Listing 5 and Listing 6 show a snippet of the HDTs for ROS 1 vs. ROS 2 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). These are the fields that FIRM generates in the template configuration (Listing 4). Comments can be written by prefixing an exclamation sign to the tag (i.e., line 1 in Listing 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 ~ (i.e., line 3 or 6 in Listing 6) are present. They are used conditionally to generate the code contained 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 6.

ROS 2 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 and Listing 6 and how they can be easily expanded highlight the benefits of our approach and how Challenge 3: Adaptivity was tackled. Small changes to include the field padding in the intermediate message model and a complementary set of HDTs (extended from the ones for ROS 1) were only needed to provide support also for ROS 2. Table 3 shows the difference of lines of code for both sets of HDTs. Note that each set of templates is composed by multiple files (called partials) to modularize their design, improving the maintainability.

As expected from the snippets shown in Listing 5 and Listing 6, the HDTs for ROS 2 have more lines of code than the ones for ROS 1. Not only the hardware components for publishers (msg to AXIS) and for subscribers (AXIS to msg) are modified. The template to generate the stimulus is also updated, mainly due to new built-in types included in ROS 2.

C. 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-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 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 it 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) [7], which offers a large set of different functions that can be combined in a “building blocks” fashion. The ROS msg specification chosen is sensor_msgs/Image (Listing 7), a complex one because it includes different data types, constrained and unconstrained sized variables. The process begins by writing the Configuration File (Listing 1, Line 1), which is the input of the workflow. Specific details of the platform (lines 2 to 6) are needed. Then, follows the information related to the accelerators to be interfaced. FIRM takes it as its input, as well as the ROS msg specifications listed there to generate the required interfaces. Additionally, TCL scripts for Vivado to build the whole project including the

![FIGURE 11: Amount of ROS messages with and without nested messages](image)
components shown in Figure 1 are generated. A current limitation of our approach is that the hardware accelerators need to be designed and added manually after the project has been built.

**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 msg specification used here is `geometry_msgs/Twist` as it is composed by linear and angular speeds in three axes. Our workflow allows to seamlessly generate a new hardware architecture by only modifying the configuration file (Listing 1), specifying the platform (Lines 2 to 6) and the name of the ROS msg 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 has to be specified, as shown in Listing 1. Zynq-7000 or Ultrascale are the supported ones (extendable if needed) and the FPGA itself as well as board can be any model.

Both experiments rely on FreeRTOS (providing an off-the-shelf TCP/IP stack) running on one ARM core. A Direct Memory Access (DMA) is used to exchange data between PS-PL imposing a bandwidth limitation. Its clock frequency is set to 300MHz and taking into account the design decision of using 8 bits for data transfers, the maximum achievable throughput is 2.4Gb/s. The Manager adds a latency of 2 clock cycles, needed to extract an ID embedded in each AXIS frame to route them accordingly. Despite of this, experiments showed that almost 50fps for 1920x1080 image resolution can be achieved if needed (Table 4). Therefore, the accelerator-related and message-dependent components do not bring significant overhead in terms of execution time for the hardware accelerators. However, it is possible to achieve a larger bandwidth if needed by increasing `tdata`’s width (up to 256 bits). The overhead introduced by the autogenerated components in terms of resource utilization and performance is evaluated (Table 5) as well as the advantages of our automation workflow compared to the traditional one.

**Resource Utilization**

Table 6 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 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 (DSPs usage). Table 5 shows the execution time of the Sobel filter combined with the autogenerated components compared to the execution of only the accelerator by HLS. Regardless of the image resolution, the increment is due to the latency introduced by the Manager which can be considered negligible.

**Automation**

Listing 1 shows how 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 specifications

![Diagram](image)

**Figure 12: Image processing use case sequence**

| Resolution      | Execution time [ms] | Frames per second |
|-----------------|---------------------|-------------------|
| 640x480         | 3.07                | 325               |
| 800x600         | 4.80                | 208               |
| 960x720         | 6.91                | 144               |
| 1024x768        | 7.86                | 127               |
| 1280x720        | 9.21                | 108               |
| 1920x1080       | 20.73               | 48                |

**Table 4: Frames per second achieved with the image processing use case**
TABLE 5: Execution time with and without generated components

| Resolution | Execution time for the accelerator [ms] | Execution time with generated components [ms] |
|------------|----------------------------------------|---------------------------------------------|
| 640x480    | 3.07                                   | 3.08 (+0.25 %)                             |
| 800x600    | 4.80                                   | 4.81 (+0.20 %)                             |
| 960x720    | 6.91                                   | 6.94 (+0.40 %)                             |
| 1024x768   | 7.86                                   | 7.88 (+0.19 %)                             |
| 1280x720   | 9.21                                   | 9.23 (+0.15 %)                             |
| 1920x1080  | 20.73                                  | 20.76 (+0.11 %)                            |

TABLE 6: Resource utilization for both use cases

|                                | Image Processing (Zynq UltraScale+) | FPGA-based Mobile Robot (Zynq 7000) |
|                                | 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   |

VI. CONCLUSION

This work presents a model-based approach to automatically generate hardware components for FPGAs to handle the compute-intensive tasks for robotic applications. A simple specification of the expected system is only needed to generate the hardware architecture. Hardware components acting as interfaces for the accelerators are obtained from message specifications, supporting arbitrary ROS messages. An intermediate message representation is obtained from the input specification, detailing the type of interface required. This, in addition to a set of templates, derived from a HDL module, are used to automatically generate the corresponding hardware components to interface the accelerators.

An evaluation of all message specifications included in both ROS versions (all latest distributions) was performed. 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 codes 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.

Future work will consist of exploring the extensibility of the toolchain to automate the insertion of accelerators and add DPR into the model-driven workflow.

6It can include multiple message specifications, publishers, and subscribers.
REFERENCES

[1] A. Martín, B. Paul, C. Jianmin, C. Zhifeng, D. Andy, D. Jeffrey, D. Matthieu, G. Sanjay, I. Geoffrey, I. Michael, et al., “Tensorflow: A system for large-scale machine learning,” in Proc. Symposium on Operating Systems Design and Implementation (OSDI), (Savannah, GA), pp. 265–283, Nov. 2016.

[2] O. Ulusel, C. Ricardo, C. B. Harris, S. Reda, and R. I. Bahar, “Hardware acceleration of feature detection and description algorithms on low-power embedded platforms,” in Proc. Int. Conf. on Field-Programmable Logic and Applications (FPL), pp. 1–9, IEEE, 2016.

[3] S. Venieris and C.-S. Bouganis, “fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs,” in Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM), pp. 40–47, IEEE, 2016.

[4] I. Kwon and J. Rose, “Measuring the gap between FPGAs and ASICs,” IEEE Transactions on computer-aided design of integrated circuits and systems, vol. 26, no. 2, pp. 203–215, 2007.

[5] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC,” in Proc. Int. Conf. on Field-Programmable Logic and Applications (FPL), 2016.

[6] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, “Angel:Eye: A complete design flow for mapping CNN onto embedded FPGAs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 1, pp. 35–47, 2017.

[7] L. Kalms, A. Podlubne, and D. Gühringer, “HiFlipVX: An Open Source High-Level Synthesis FPGA Library for Image Processing,” in Proc. Symposium on Applied Reconfigurable Computing (ARC), 2019.

[8] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al., “Going deeper with embedded fpga platform for convolutional neural network,” in Proc. Int. Symposium on Field-Programmable Gate Arrays (FPGA), pp. 26–35, 2016.

[9] J. Qiu, W.-L. Chao, D. Garg, B. Harilaran, M. Campbell, and K. Q. Weinberger, “Pseudo-LiDar From Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving,” in Proc. Int. Conf. on Computer Vision and Pattern Recognition (CVPR), 2019.

[10] Y. Hao, M. Carraro, M. Munaro, and E. Menegatti, “Robust multiple object tracking in rgb-d camera networks,” in Proc. Int. Conf. on Intelligent Robots and Systems (IROS), 2017.

[11] S. Venieris, S. Tamura, and H. Takase, “A Study on Introducing FPGA to ROS Based Autonomous Driving System,” in Proc. Int. Conf. on Field-Programmable Technology (FPT), 2018.

[12] K. Hasegawa, K. Takasaki, M. Nishizawa, R. Ishikawa, K. Kawamura, Y. Nitta, S. Tamura, and H. Takase, “A Study on Introducing FPGA to ROS Based Autonomous Driving System,” in Proc. Int. Conf. on Field-Programmable Technology (FPT), 2019.

[13] Y. Nitta, S. Tamura, H. Yugen, and H. Takase, “ZytleBot: FPGA integrated development platform for ROS based autonomous mobile robot,” in Proc. Int. Conf. on Field-Programmable Technology (FPT), 2019.

[14] J. P. Queralt, F. Yuhong, L. Saloma, L. Qingting, T. N. Gia, Z. Zou, H. Tengshen, and T. Westerland, “FPGA-based Architecture for a Low-Cost 3D Lidar Design and Implementation from Multiple Rotating 2D Liders with ROS,” in IEEE Sensors, pp. 1–4, IEEE, 2019.

[15] Aldegeri et al., “A framework for the design and simulation of embedded vision applications based on OpenVX and ROS,” in Proc. Int. Symposium on Circuits and Systems (ISCAS), 2018.

[16] K. Yamashita, T. Ohkawa, K. Ootsu, and T. Yokota, “Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems,” CoRR, vol.abs/1508.07123, 2015.

[17] T. Okhawa, Y. Sugata, H. Watanabe, N. Ogura, K. Ootsu, and T. Yokota, “High level synthesis of ROS protocol interpretation and communication circuit for FPGA,” in Proc. Int. Workshop on Robotics Software Engineering (RoSE), 2019.

[18] D. P. Leal, M. Sugaya, H. Amano, and T. Ohkawa, “FPGA Acceleration of ROS2-Based Reinforcement Learning Agents,” in Proc. Int. Symp. on Computing and Networking Workshops (CANDARW), pp. 106–112, IEEE, 2020.

[19] M. Eisoldt, S. Hindrink, M. Tassemerle, M. Flottmann, J. Vana, T. Wiejmann, J. Gaal, M. Rothmann, and M. Porrmann, “ReconROS: Running ROS on Reconfigurable SoCs,” in Proc. on Workshop on Drone Systems Engineering (Drose), 2021.

[20] C. Lienen, M. Platzner, and B. Rinner, “ReconROS: Flexible Hardware Acceleration for ROS2 Applications,” in Proc. Int. Conf. on Field-Programmable Technology (FPT), 2020.

[21] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzer, and C. Plessl, “ReconOS: An operating system approach for reconfigurable computing,” IEEE Micro, vol. 34, no. 1, pp. 60–71, 2013.

[22] F. M. Costa, K. A. Morris, F. Kon, and P. J. Clarke, “Model-driven domain-specific middleware,” in Proc. Int. Conf. on Distributed Systems (ICDCS), pp. 1961–1971, IEEE, 2017.

[23] J. Wienke, A. Nordmann, and S. Wrede, “A meta-model and toolchain for improved interoperability of robotic frameworks,” in Simulation, Modeling, and Programming for Autonomous Robots, (Berlin, Heidelberg), pp. 323–334, Springer Berlin Heidelberg, 2012.

[24] A. Podlubne and D. Gühringer, “FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs,” in Proc. Int. Conference on ReConfigurable Computing and FPGAs (ReConFig), pp. 1–5, IEEE, 2019.

[25] G. Metta, P. Fitzpatrick, and L. Natale, “YARP: yet another robot platform,” International Journal of Advanced Robotic Systems, vol. 3, no. 1, p. 8, 2006.

[26] H. Bruyninckx, “Open robot control software: the OROCOS project,” in Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164), vol. 3, pp. 2523–2528, IEEE, 2001.

[27] D.C. Schmidt, “Model-driven engineering,” Computer-IEEE Computer Society.

[28] E. N. Koth, “Semantics of context-free languages,” Mathematical systems theory, vol. 2, no. 2, pp. 127–145, 1968.

[29] R. Farrow, “Generating a Production compiler from an Attribute Grammar,” IEEE Software, vol. 1, pp. 77–93, Oct. 1984. Publisher: IEEE Computer Society.

[30] C. Bürger, S. Karol, and C. Wende, “Applying attribute grammars for metamodel semantics,” in Proceedings of the International Workshop on Formalization of Modeling Languages, p. 1, ACM, 2010.

[31] J. Oqvist, “ExtendI: extensible Java compiler,” in Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Programming’18 Companion, (New York, NY, USA), pp. 234–235, Association for Computing Machinery, Apr. 2018.

[32] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, “Higher order attribute grammars,” in ACM SIGPLAN Notices, vol. 24, no. 3, pp. 131–145, ACM, 1989.

[33] J. Mey, R. Schön, G. Hedin, E. Söderberg, T. Kühn, N. Fors, J. Oqvist, and U. Almann, “Relational reference attribute grammars: Improving continuous model validation,” Journal of Computer Languages, vol. 57, 04 2020.

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

[35] J. Mey, T. Kühn, R. Schön, and U. Assmann, “Reusing static analysis across different domain-specific languages using reference attribute grammars,” The Art, Science, and Engineering of Programming, vol. 4, no. 3, pp. 15:1–36, 2020.

[36] C. Wansstrath, “Mustache - logic-less templates.” https://mustache.github.io, 2020. Accessed: 2020-07-20.

[37] S. Mittapalli and M. P. Arthur, “Survey on Template Engines in Java,” ITM Web of Conferences, vol. 37, p. 01007, 2021. Publisher: EDP Sciences.
ARIEL PODLUBNE is a research assistant and PhD student at the Chair of Adaptive Dynamic Systems at the Technische Universität Dresden. He graduated as an MSc. Electrical Engineer in December 2012 from the National Technological University in Cordoba, Argentina. He specialized in Digital Electronics and Embedded Systems during his studies. He received the “Binid” scholarship in 2013, which is for freshly graduates to start a research career. His workplace was CINTRA UA-CONICET where he designed and developed an FPGA-based tool to optimize heterogeneous data acquisition and processing, in order to conduct tests on human echolocation. He worked for two years at LAAS/CNRS in Toulouse France from June 2014 for the European project Two/Ears developing the robotics testbed used for experiment. His research interests are FPGA-Based architectures for signal and image processing and robotics.

JOHANNES MEY is a research assistant and PhD student at the Chair of Software Technology at Technische Universität Dresden. His research focuses on reference attribute grammars and their application in various fields. Besides static program analysis, these include model transformations and adaptive software systems. Furthermore, he investigates the relation between reference attribute grammars and conceptual models.

RENÉ SCHÖNE is a research assistant and PhD student at the Chair of Software Technology at Technische Universität Dresden. His research and PhD focuses on the application of reference attribute grammars for models@runtime currently within the domain of smart home. Challenges there include adequate modelling of domains, abstraction of and connection to real hardware devices, and efficient analyses in the present of frequent model updates.

DIANA GÖHRINGER holds the Chair of Adaptive Dynamic Systems at the Technische Universität Dresden. From 2013 to 2017 she was an assistant professor and head of the MCA (Application-Specific Multi-Core Architectures) research group at Ruhr-University Bochum (RUB), Germany. Before that she was working as the head of the Young Investigator Group CADEMA (Computer Aided Design and Exploration of Multi-Core Architectures) at the Institute for Data Processing and Electronics (IPE) at the Karlsruhe Institute of Technology (KIT). From 2007 to 2012, she was a senior scientist at the Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB in Ettlingen, Germany (formerly called FGAN-FOM). In 2011, she received her PhD (summa cum laude) in Electrical Engineering and Information Technology from the Karlsruhe Institute of Technology (KIT), Germany. She is author and co-author of 1 book, 10 invited book chapters and over 130 publications in international journals, conferences and workshops. Additionally, she serves as technical program committee member in several international conferences and workshops. She is reviewer and guest editor of several international journals. Her research interests include reconfigurable computing, multiprocessor systems-on-chip (MPSoCs), networks-on-chip, simulators/virtual platforms, hardware-software codesign and runtime systems.

UWE AßMANN holds the Chair of software technology at the Technische Universität Dresden. He has obtained a PhD in compiler optimization and a habilitation on invasive software composition (ISC), a composition technology for code fragments enabling flexible software reuse. ISC unifies generic, connector-, view-, and aspect-based programming for arbitrary program or modelling languages. Since 2013, he is deputy of the DFG Research Training Group Role-oriented Software Infrastructures (RoSI), which develops new techniques for context-adaptive software, from language and application design to runtime (rosi-project.org).