IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 September 2023, accepted 5 October 2023, date of publication 18 October 2023, date of current version 25 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3325742

RESEARCH ARTICLE

Enabling Plug-and-Play in Cyber-Physical Systems Using MPSoC-FPGAs

DANIELE PASSARETTI[1], (Graduate Student Member, IEEE), MAX STEIGER[2], AND THILO PIONTECK[1], (Member, IEEE)

[1]Institute for Information Technology and Communications, Otto-von-Guericke University Magdeburg, 39106 Magdeburg, Germany
[2]Department of Simulation and Graphics, Otto-von-Guericke University Magdeburg, 39106 Magdeburg, Germany

Corresponding author: Daniele Passaretti (daniele.passaretti@ovgu.de)

This work was supported in part by the Federal Ministry of Education and Research within the Project KIDs-CT under Grant 13GW0229A, and in part by the Open Access Publication Fund of Magdeburg University.

ABSTRACT Cyber-Physical Systems (CPSs) combine computation, networking, and physical processes. A CPS consists of various independent subsystem components with different interfaces and protocols defined by vendors. Utilizing different interfaces and protocols with the necessity for flexible, dependable, and extensible CPSs poses new challenges for controlling, communicating, and synchronizing the components. In this paper, we propose a Centralized Control Unit (CCU) for CPSs and the associated Communication Infrastructure that enable the support of subsystem components as ''plug-and-play'' modules. Our CCU consists of a hardware-software architecture that handles physical signals as well as real-time and non-real-time tasks implemented on a single MPSoC-FPGA. To enforce the dependability and flexibility of the communication between components of different vendors, we model the Communication Infrastructure in layers with communication classes. In addition, we propose three vendor-agnostic application protocols, which are also implemented in the CCU. To validate our work, we have implemented and integrated the CCU with the communication infrastructure into an open-interface Computed Tomography (CT) scanner, where sensors/actuators can be added in a ''plug-and-play'' fashion. We have also evaluated the effort of integrating an additional detector into our scanner. The CCU runs on an AMD-Xilinx Zynq-7000 XC7Z045, and it uses only 10% of the hardware resources.

INDEX TERMS Computed tomography, cyber-physical systems, MPSoC FPGA, centralized control unit, industry 4.0.

I. INTRODUCTION

Cyber-Physical Systems (CPSs) are the core of the new industrial revolution called ''Industry 4.0'' [1], [2], [3]. They are widely used in many application areas: automotive systems, medical instruments, and aerospace control technologies. In the medical field, CPSs are used in various categories of devices [4], [5], [6]. Medical CPSs include small implantable items such as mobile devices for monitoring and alarming, and heavyweight stationary systems such as Computed Tomography (CT), Positron Emission Tomography (PET), or Magnetic Resonance Imaging (MRI) systems [7].

At the device level, a CPS consists of components that control processes while taking care of physical and computational requirements like sensitivity, stability, reliability, dependability, speed, and noise reduction [8]. To achieve the proposed capabilities at the device level, CPSs utilize a management control system with different control units for the various components (e.g., the physical sensor/actuator and the control processing system) [9]. These control units are connected to the main control unit that can be centralized, distributed, or decentralized along edge nodes, sensors, and actuators on single or multiple chips [10].

A. MOTIVATION

In recent years, the rising number of sensors/actuators and the integration of autonomous and intelligent systems into CPSs have increased the complexity of control, synchronization, and data processing algorithms [11]. In order to meet all requirements in real-time, CPS vendors tend to use

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

solutions based on Centralized Control Units (CCUs) that are implemented on a single chip like Multi-Processor System-on-Chips Field Programmable Gate Array (MPSoC-FPGAs). Although these hardware platforms can fulfill the implementation requirements of CPSs, they open new challenges in terms of interoperability, flexibility, and scalability of the CPS [12], [13]. For example, Hoffman [14] highlights that easy and flexible integration of sensors will be crucial for the widespread acceptance of CPS concepts. These challenges, which include the communication and interoperability of components and the control unit of the CPS, are often addressed in the literature as ''plug-and-play'' capability [15], [16], [17], [18], [19].

To address these challenges in medical CPSs, the Medical Device Plug-and-Play (MDPnP) initiative has been promoted, which aims to create new models, architectures, and standards to support plug-and-play extensions to the system [17], [18], [19].

B. CONTRIBUTION

This paper proposes a Communication Infrastructure with associated protocols and a CCU architecture for MPSoC-FPGAs, which provide the plug-and-play capability, considering real-time requirements. These facilitate the interoperability and integration of CPS components. In contrast to previous works, the proposed Communication Infrastructure supports components with standard and custom protocols. In order to provide the ''plug-and-play'' capability without affecting latency and dependability, we have modeled our Communication Infrastructure in three layers that group communication interfaces, transport, and application protocols. Each layer has three classes: non-real-time, real-time control/synchronization, and real-time data classes that allow to fulfill the different task requirements. In addition, we have proposed an application protocol for each class, which permits synchronization and inter-communication of components at the application protocol layer.

The CCU is a node instance of the Communication Infrastructure. To design the CCU architecture, we have used a hardware/software co-design methodology, where real-time protocols and application tasks are implemented as hardware modules and non-real-time tasks are implemented as software modules. To enable the ''plug-and-play'' feature in the CCU and to have the possibility to integrate new hardware modules, we use an MPSoC-FPGA, which is a reconfigurable device.

For validation and evaluation purposes, we have used the Communication Infrastructure and the CCU to interconnect and control components of an open-interface Computed Tomography (CT) scanner. This is an example of an MDPnP CPS [17] where it is possible to add Detector Management Systems (DMSs), X-ray tubes, and collimators in a plug-and-play fashion. This plug-and-play feature also impacts medical aspects because it facilitates the exploration of new clinical acquisition techniques such as multi-modality imaging [20]. In addition, it permits updating the components in the CT scanner, which is an open challenge pursued by the high cost of CT scanners [21]. Moreover, thanks to the CCU architecture, the components can be synchronized in real-time in the order of a few clock cycles, and the whole open-interface CT can collect and process data in real-time, which is essential for medical intervention applications [22], [23].

Finally, the CCU architecture with the communication infrastructure should lay a base framework for further research or implementation projects, providing a starting point for implementing complex CPSs with few constraints at the device level.

C. STRUCTURE

The rest of this paper is organized as follows: Sec. II reports the related work on CCU architectures and medical CPSs system architectures. Sec. III describes the proposed Communication Infrastructure and the Application protocols. Sec. IV presents the software-hardware architecture of the CCU. Sec. V presents the implementation of the CCU and the Communication Infrastructure for the open-interface CT. Sec. VI discusses the results of the CCU and the Communication Infrastructure on the running open-interface CT, compared with state-of-the-art performance and features.

II. RELATED WORKS

The design of cyber-physical systems that can provide interoperability and flexibility between sensors and actuators is a well-known problem in the literature at [2], [12], and [24]. To solve this problem at the system and communication level, the National Institute of Standards and Technology (NIST) has established the Cyber-Physical Systems Public Working Group (CPS PWG) [12]. It is an open public forum that supports designers and aims to develop a generic framework for cyber-physical systems. It starts by considering the three facets of CPSs: conceptualization, realization, and assurance. In relation to them, the framework proposes activities for defining the different CPS models. During these steps, it points out the interoperability problems between different components in the CPS caused by the incompatibility of the different protocols used. In his literature review, Hofer [2] also identifies component interoperability as a challenge in the different analyzed CPS architectures.

While these previous works point out the interoperability and extensibility problem of CPS and show that the communication infrastructure and the control unit are the key elements to address this problem, they do not propose any software/hardware architecture at the device level. However, several solutions have also been proposed at the CPS device level to solve such a problem [25], [26], [27]. These show that besides the different domains and their applications to provide interoperability and extensibility, a major challenge at the device level should be solved: designing control units that provide plug-and-play capability for vendor components that use custom and standard protocols. The various solutions

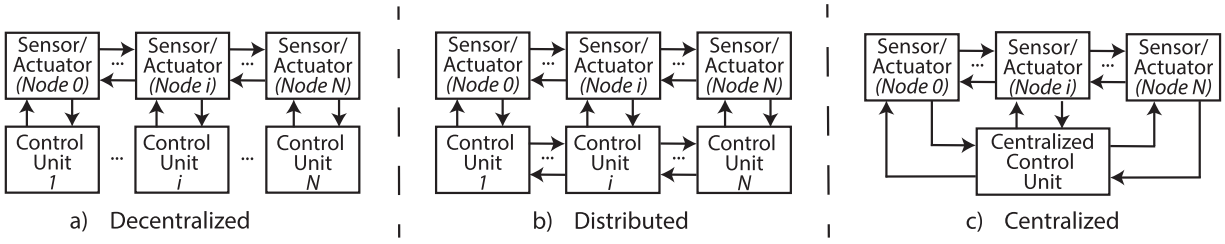

FIGURE 1. Control unit models.

for designing control units are based on a Centralized, a Distributed, and a Decentralized architecture [10], as shown in Fig. 1.

In CPSs, which process a huge amount of data in real-time, the centralized solution is mainly used. For instance, Tesla and AMD-Xilinx have introduced a Full Self-Driving (FSD) hardware architecture for Advanced Driver-Assistance Systems (ADAS) [25], [26]. Both are examples of dedicated CCU architectures for CPSs with a dedicated hardware architecture, the first targeting Application Specific Integrated Circuits (ASICs), and the second being synthesized on MPSoC-FPGAs [28]. Although these FSDs collect and acquire data from different sensors/actuators in real-time, they are not extensible in a plug-and-play fashion.

Also, in the medical field, various CPS architectures have been proposed for CT, MRI, or similar medical devices [2], [5], [27], [29], [30], [31], [32], [33]. Reference [29] describes an architecture for CT that uses a distributed control unit, also proposed by AMD-Xilinx. It is divided into three components: the "*System Sequencer*", the "*High Voltage Supply Control*", and the "*Data Acquisition & Gantry Control*" systems. The System Sequencer is responsible for synchronizing the various components. The High Voltage Supply Control monitors and controls the voltage for the X-ray tube and the detectors; this implements safety-critical tasks that regulate the X-ray dose. The Data Acquisition & Gantry Control collects data and controls the DMS and the gantry. This distributed approach avoids the problem of isolating different tasks and communication interfaces. However, it is not scalable, and components cannot be added in a plug-and-play fashion. In fact, in order to integrate a new component (e.g., detector or X-ray tube), tasks must be divided into subtasks and mapped across the distributed control units. This problem is discussed in Sec. VI, where we also compare our work with the AMD-Xilinx solution.

In PET applications, Min et al. [31] developed a low-cost CPS using an FPGA as a CCU. In this system, the DMS can be configured before each scan to acquire and process the data before transmitting them to the host PC. Due to the speed limitations of the USB connection between the acquisition system and the host PC, it can only be used for applications that reconstruct the image offline. The system is optimized for the specific application, using hardware modules without a software stack that limits future extensions.

In a similar way, Korcyl et al. [32] define a custom architecture with a CCU for a configurable PET system. The CCU controls the DMS using a configuration set that is sent by the host PC. Furthermore, the CCU processes and transmits the acquired data via an optical link. The CCU can only be reprogrammed for the different PET configurations and cannot be used for other CPS applications.

A more generic CPS architecture for image acquisition is proposed by Fysikopoulos et al. [33] as a customized solution for nuclear medicine applications. The architecture uses an FPGA as a CCU that can be reprogrammed to use other DMSs. In order to communicate, this CCU uses Ethernet with UDP and a custom datagram protocol for data acquisition.

Most of the related works in [2], [5], [30], [31], [32], and [33] present custom CCU architectures where the CPS itself is only exploited vertically from the cyber to the physical level. They do not consider the Communication Infrastructure, its implications on the CCU implementation, and the problem of having a universal generic CPS at the device level. In fact, they propose architectures for specific sensors/actuators for the targeted application, without considering the interoperability and extensibility issues. These CPS challenges are highlighted by Liu et al. [27], who propose to shift the research to a more general proposal of frameworks for a wide range of applications.

III. COMMUNICATION INFRASTRUCTURE

In this section, we describe our Communication Infrastructure, which is the glue between the different subsystem components. It defines the internal communication of the CPS at the device level and defines the component interfaces as well as the transport and application protocols. Furthermore, the Communication Infrastructure aims to be vendor-agnostic and to provide plug-and-play support for new sensors/actuators. In contrast to related works, the proposed communication infrastructure is not limited to a standard protocol, but it can also be implemented with custom interfaces and protocols. For example, components of the targeted medical application domain often use custom protocols, and the Open Platform Communications Unified Architecture (OPC UA) cannot be used for these custom protocols.

To achieve a Communication Infrastructure that is vendor-agnostic and provides plug-and-play capability for new sensors/actuators, we have defined components as nodes and modeled them with layers and classes, as shown in Fig. 2

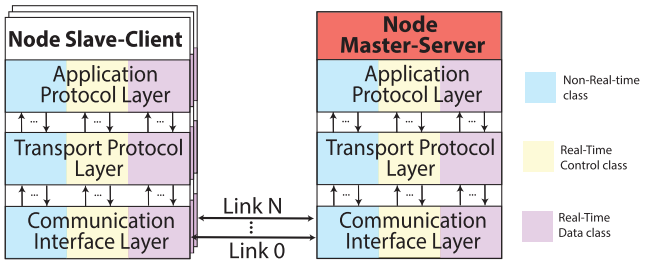

FIGURE 2. Communication infrastructure.

and in Fig. 3. There are two types of nodes: the *master-server node* and the *slave-client node*. All slave-client nodes are connected to the master-server node. This centralized solution facilitates task interactions for real-time requirements. Each node has three layers: the *communication interface layer*, the *transport protocol layer*, and the *application protocol layer*. This layered organization is the key element upon which the proposed plug-and-play solution is based. For managing the different types of interfaces, protocols, and tasks per layer, we have defined different classes per layer, which are the *real-time control class*, the *real-time data class*, and the *non-real-time class*. As shown in Fig. 2, these classes are based on their common timing and dependable communication properties. Using different classes for the various task types permits the designer to easily associate the protocols of new components with the proper application protocol. In addition, it permits the optimization of the deadline in relation to the task properties.

The layered architecture with classes permits the designer to add components to the CPS as plug-and-play modules. In fact, when a new component is added to the CPS, a new node with associated interfaces and protocols is defined in the Communication Infrastructure. Based on the node's communication requirements, interfaces, and protocols are associated with the appropriate class for each layer.

At the communication interface and transport protocol layers, we consider the various interfaces and protocols provided by vendors. At the application protocol layer, we propose a component-agnostic protocol per class, unifying the different vendor protocols and enforcing them in the CCU. An example of a single node with different protocols (UART, IPv4, etc.) is shown in Fig. 3, where the proposed protocols for the application protocol layer are written in red. In the following paragraphs, we describe the types of *node*, the interfaces, and the protocols involved in the Communication Infrastructure.

A. NODES

Nodes represent independent subsystem components of the CPS (e.g., sensor, actuator, controller, and data processing units). The Communication Infrastructure has two types of nodes: the master-server node and the slave-client node.

The **master-server node** controls and synchronizes slave-client nodes. It implements the server used for non-real-time communication and the masters used for master-slave real-time communication. Due to the heterogeneity of the tasks and protocols, the CCU that implements this node is designed to be implemented on an MPSoC-FPGA. This platform allows easy integration of new custom interfaces and protocols that can be implemented on the Programmable Logic (PL) and controlled by tasks implemented on the Processing System (PS).

The **slave-client node** is responsible for controlling the onboard sensors/actuators or processing tasks. Each slave-client node has its own local control unit. In this way, in the event of communication failures, each slave-client node runs autonomously in safe mode without propagating errors. Slave-client nodes correspond to internal components, and they usually are provided by one or more vendors. For this reason, they may use different protocols and interfaces that cannot communicate directly with each other. Depending on the functionality of the task, they can be implemented using a workstation PC, a microcontroller, an MPSoC, or an FPGA.

To be compatible with the Communication Infrastructure, a component must have at least one control interface, which can be real-time or non-real-time; nodes cannot have only the *real-time data interface*, otherwise they cannot be controlled or triggered. For example, in Fig. 4, we show an example with the possible node cases.

In Fig. 4, nodes 3 and 4 have only one interface, which is the non-real-time and the real-time control interfaces, respectively. Both can be controlled and synchronized by the CCU. In fact, if these nodes need to communicate with each other, the CCU handles the respective communication tasks at the application level.

B. COMMUNICATION INTERFACE LAYER

As mentioned, each node may have one or more interfaces with different timing and dependability requirements. For this reason, we propose the following three classes of communication interfaces, as shown in Fig. 3.

1) NON-REAL-TIME INTERFACE CLASS

This class is used for the communication between non-real-time tasks of different nodes (e.g., asynchronous setting-up of sensors/actuators, data logging, and asynchronous control tasks). The key idea behind this class is to group all interfaces that will use the client-server class in the upper layer. Examples of interfaces are Ethernet and Wi-Fi connections. Although these interfaces have different data rates, they use the same class in the protocol layer.

2) REAL-TIME CONTROL INTERFACE CLASS

This class handles control signals, synchronizes nodes, and communicates between real-time control tasks of different nodes. Interfaces of this class support real-time communication. Examples of instances for this communication interface class are the interfaces of UART, SPI, I2C, EtherCAT protocols, and/or custom protocols using custom signals. All of these interfaces use master-slave protocols.

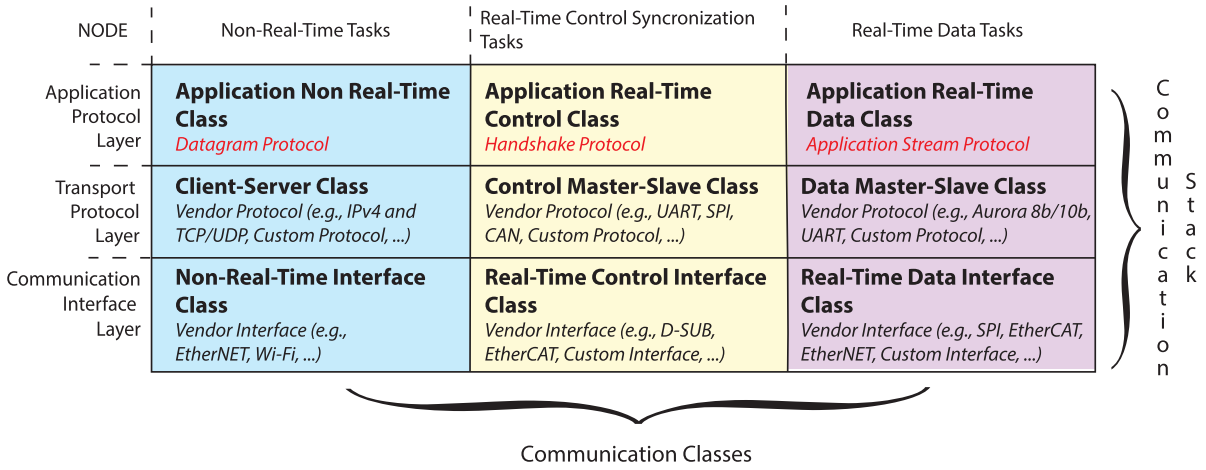

FIGURE 3. Model of the node in the communication infrastructure.

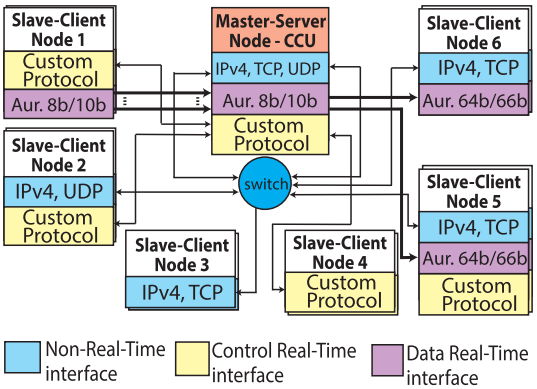

FIGURE 4. Example of node interconnection and interface layer.

3) REAL-TIME DATA INTERFACE CLASS

This class groups interfaces associated with real-time dataflow tasks, where large amounts of data are transmitted from sensors to the collecting system. To support high-speed communication, the transceiver can require a dedicated reference clock. For this reason, interfaces of this class are usually mapped with transceivers, which are placed on the PL part of the MPSoC-FPGA. The key idea behind this class is to provide support for master/slave communication in stream fashion on the upper layer, independently by its data rate.

C. TRANSPORT PROTOCOL LAYER

On top of the *Interface layer*, there is the transport protocol layer. Depending on time and dependability requirements, we have grouped the different protocols into three classes:

1) CLIENT-SERVER CLASS

This class contains client-server protocols for non-real-time communication. It includes all protocols from the *Data link layer* up to the *Transport layer* of the OSI model stack (e.g., IPv4 and TCP/UDP" [34]). The key idea behind this class is to group protocols that are unified in the *application protocol layer*.

2) CONTROL MASTER-SLAVE CLASS

This class considers protocols where data or commands are transmitted for real-time controlling/synchronization purposes. In fact, there aren't any real-time protocols in this class where a large amount of data needs to be transmitted. Instead, the main requirement is the dependability of the communication. For this reason, we also propose a Handshake Protocol on the application level. Examples of protocols in this class are UART, I2C, SPI, and Ether-Cat. Also, custom protocols that use control signals are encapsulated in packets at this level.

3) DATA MASTER-SLAVE CLASS

This class includes protocols where data are transmitted in real-time in stream mode. The transceivers implementing these protocols also use a stream protocol within the node architecture (e.g., AMBA 4 AXI4-Stream protocol [35]). This allows data to be sent/collected and processed on-the-fly, which is essential for the application layer. Examples of protocols in this class are Aurora 8b/10b and PCI-Express configured in stream mode.

D. APPLICATION PROTOCOL LAYER

On top of the communication stack, as shown in Fig. 3, we have the application layer, which also consists of three classes: non-real-time, real-time control/synchronization, and data class. In contrast to the previous layers, where protocols are usually defined by vendors, for this layer we have proposed a single protocol per class. With a unique protocol per class node, it is possible to exchange information between nodes with different interfaces and transmission protocols, and it is possible to provide plug-and-play support for new components.

1) APPLICATION NON-REAL-TIME CLASS

This class groups non-real-time tasks that use client-server communication. Messages for these types of tasks can range

in size from a few bytes to several megabytes. To unify them, we have proposed the **Application Datagram Protocol**. In this protocol, data are sent in messages that are variable in size and flexible in the data format to be sent.

Each message encapsulates from 0 to N data packets, and each packet has its own variable size and data format. In this way, commands and data packets can be encapsulated in two packets of a single message. As shown in Fig. 5, the application datagram protocol has three segments: header, data, and tail. The sizes of the internal fields are multiples of 16 bits. If a command is smaller than 16 bits, spare bits are added during the transmission.

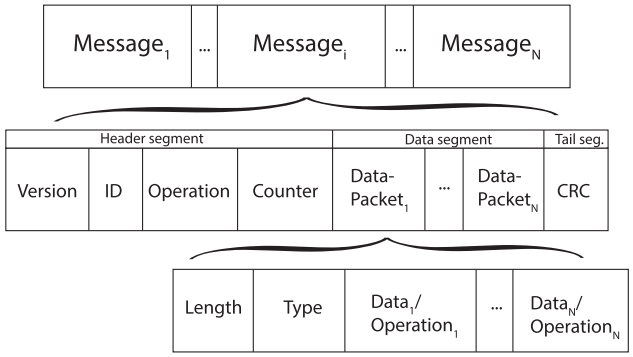

FIGURE 5. Message structure of the application datagram protocol.

The **Header segment** has four mandatory fields, as shown in Fig. 5. The *Version* field defines the client-server protocol version in use. If a client sends a request message with a different protocol version than the server, the communication is closed. Each request or application message has a *Unique Identifier* (UID); if the size of an application message exceeds the Maximum Transmission Unit, it is split into multiple messages, and the UID is incremented by 1. The *Operation* field specifies the operation associated with the message. In relation to the slave-client node, each operation has a number of commands and/or task operations. This number is specified by the field *Counter*; if it is equal to 0, the Data segment is skipped.

The **Data segment** contains the commands and data tasks as data packets of variable size. For this reason, each of them has the fixed fields *Length* and *Type*, which specify the number of bytes and the data type of the *Data* fields. These fields allow the receiver to dynamically calculate the data packet size, the amount of *Data* fields, and their encoding. For instance, in the implemented used case, the pong message has the *Length* field of 8 and the Type of 4, so the data packet contains 2 data fields of 4 bytes each. At the end of the message, there is the **Tail segment**, which contains the Cyclic Redundancy Check (CRC) field that specifies the 32-bit CRC of the previous segments.

2) APPLICATION REAL-TIME CONTROL CLASS

This class groups real-time control/synchronization communication tasks and signals using standard or custom real-time protocols. Due to their diversity and real-time requirements, it is not possible to define a generic application protocol for them, such as the Application Datagram Protocol. Instead, we have proposed a **Handshake Protocol** on top of them, which handles eventual errors and lost messages in real-time. It is implemented in the PL part of the CCU side and connected to each transceiver instance. In this way, slave-client nodes do not require additional implementations.

The transport protocols must fulfill two requirements for supporting the Handshake Protocol. First, the communication must only start from the master side. Second, the master-slave protocol must send an acknowledgment (ACK) for each message received. If a control/synchronization message does not contain a CRC field (e.g., single signal), a separate acknowledgment signal is required (e.g., valid/error signals).

Based on these assumptions, the Handshake Protocol has the following two phases, as shown in Fig. 6:

- **Transmitting phase**: In this phase, the master initiates the communication with a slave-client node. After sending a message, it waits for the corresponding ACK. For messages sent in burst mode, it can be set to receive a single ACK after the last message or a burst of ACKs. After sending messages, the master enters the Receiving phase while the receiver waits for the message.
- **Receiving phase**: When the message arrives at the slave-client node, the Receiving phase takes place on its side. In this phase, the receiver checks the integrity of the message and sends back the ACK. The ACK contains the ID of the received message, depending on the specific transmission protocol. If the ACK arrives on the master side within the timeout and the ID contained in the ACK message matches with the corresponding sent message, the communication is successfully completed, as shown in Fig. 6. In the case shown in Fig. 7, where first the master does not receive the ACK and then the ID of the received ACK does not match, a lost message and an error message are detected, respectively. In this scenario, the master automatically retransmits the original message for a maximum number of retransmissions. If no correct ACK is received beyond the maximum number of retransmissions, an error flag is set in the corresponding status register. Depending on the severity, the error event is propagated to the related software module that handles it (e.g., the CPS runs in safe mode).

To manage these errors in real-time and avoid denial-of-service attacks, the Handshake Protocol is implemented in the PL with Finite State Machines (FSMs) that control the associated transceiver. This permits the handling of timeout events and the support of a retransmission mechanism with

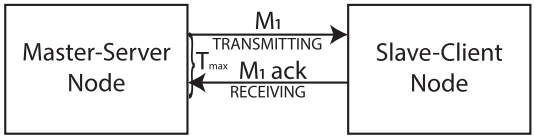

FIGURE 6. Handshake protocol in case of no errors.

appropriate signals that are physically connected to the related transceivers. In addition, if a malicious slave attempts to initiate a communication with the master, the received message is blocked within the Control-Synchronization Unit. Since all these events are handled in the PL, they do not affect the software scheduler and the execution time of the other tasks.

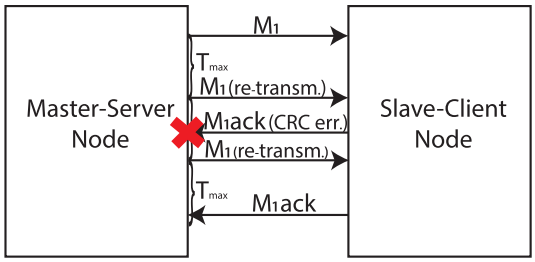

FIGURE 7. Handshake protocol. Message lost and CRC error use case.

3) APPLICATION REAL-TIME DATA CLASS

This class groups communication tasks that use the data master-slave class on the layer below. In this class, we assume serial communication protocols with a variable message size and simplex communication (i.e., one-direction channel only). Due to the lack of synchronization signals of the simplex communication, the receiver must be synchronized with the transmitter at the application layer. For these reasons, we propose an **Application Stream Protocol** that uses various commands in and between messages, as shown in Fig. 8. Commands and messages consist of packets, which are the smallest amount of information that can be sent. The packet size is fixed and set at design time.

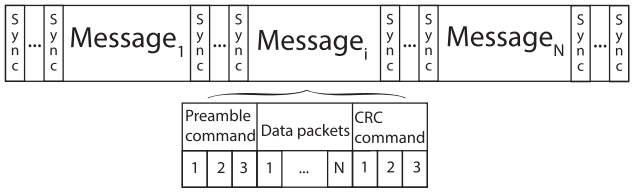

FIGURE 8. Application stream protocol.

Synchronization takes place between messages. When the CPS is powered-up, each transmitter is set to idle mode, and *synchronization commands* (i.e., packets) are sent. Packets are encoded and sent as a stream of bits that the receiver must decode and align. In this phase, the *synchronization commands* are used to match the alignment with the received message. When a message is sent, the receiver is already synchronized. A message starts with a *preamble command*, which is a sequence of three packets. After this command, the data packets are sent until the *CRC command*, which is used to identify the CRC packet. In this way, the receiver can check the integrity of the message and accept or discard it.

As shown in Fig. 8, a *synchronization command* consists of only one packet because it is sent between messages and cannot be misread as a data packet. Instead, the other commands consist of multiple packets because they are part of the message and could be wrongly misread as data packets.

IV. CENTRALIZED CONTROL UNIT

In this section, we describe the architecture of the proposed CCU, which implements the master-server node and the proposed protocols of the Communication Infrastructure. For mapping real-time and non-real-time tasks in the CCU, we have used a hardware/software co-design methodology, resulting in the multi-layer architecture shown in Fig. 9. In the CCU architecture, each layer has internally isolated modules that can only communicate with the corresponding module in the adjacent layer. Furthermore, to fulfill the predictable time execution of the real-time tasks, we implemented them in dedicated modules on the *Hardware layer*. However, we implemented the non-real-time tasks as a module in the *Application layer*. For instance, we have implemented the real-time classes of the Application Protocol layer in the CCU *Hardware layer* and the non-real-time class in the CCU *Application layer*.

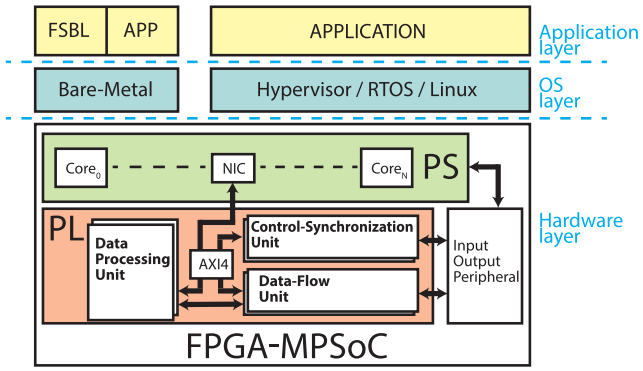

FIGURE 9. Illustration of the CCU architecture.

Software and hardware modules communicate with each other via AXI4-Lite interfaces, mapping the memory of the *Hardware layer* to the global memory space of the *Operating System* (OS), which constitutes the *OS layer*. The latter is responsible for the task scheduling and for converting the virtual address of each application into the corresponding physical address in the *Hardware layer*, guaranteeing memory isolation where it is required.

A. HARDWARE ARCHITECTURE

The **Hardware layer** consists of three main modules, implemented on the PL part, as shown in Fig. 9:

Control-Synchronization Unit: This module implements the control and synchronization tasks related to the Real-Time Control/Synchronization communications, implementing the *real-time control class*, the masters of the *control master-slave class* and the Handshake Protocol of the *application real-time control class*. To control the different nodes, the CCU uses isolated instances for each external slave node. As shown in Fig. 10, each instance has the transceivers, the *control/status registers*, the *data registers*, the *prepare-packet unit*, the *control-flow unit*, the *retransmission unit*, and

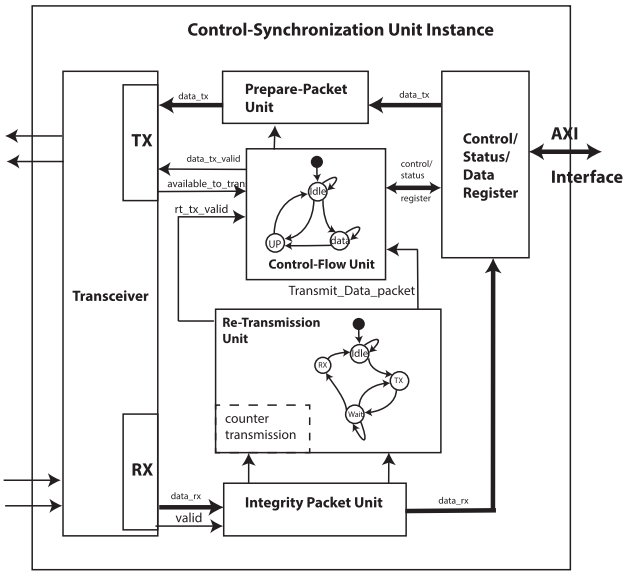

FIGURE 10. An instance of the control-synchronization unit associated with a single component.

the *integrity packet unit*. The non-real-time tasks running on the PS write data asynchronously into associated registers mapped in the global memory space through the AXI interface; these registers are also read/written in real-time by the other hardware modules of the Control-Synchronization Unit.

The *control-flow unit* checks the availability of the transceiver to send data. If it is available and there are data to send, it activates the *prepare-packet unit* and the transceiver that sends the data with its protocol. When the data is sent, the retransmission unit waits for the reply/ack from the transceiver. If no correct message has been received within the maximum time set, it enables the retransmission signal, which is connected to the control *control-flow unit*. The *retransmission unit* and the *control-flow unit* have an FSM to check the various states of this component and to generate the signals explained. The transceiver implements the transmitter/receiver for the protocol of the associated slave. For instance, in the targeted use case, the DMS uses a custom protocol, and the collimator an i2c protocol. During communication, the Integrity Packet Unit blocks any malicious messages before they reach the AXI interface.

Data-Flow Unit: This module collects data from slave-client nodes through the proposed *Application Stream Protocol*. Then, it forwards them to the Processing Unit or to slave-client nodes that are designated for Data Processing. This module is implemented in a data-flow architecture presented in our previous work [36]. It consists of independent internal pipes per slave-client node. Each internal pipe implements the transmitter/receiver and the decoder/encoder for the *Application Stream Protocol*. Since the protocol can be configured for different data rates, each pipe has its own clock domain with a fixed clock frequency depending on the data rate of the collected/forwarded data. Due to the different pipe clock frequencies, a Crossing Domain Clock (CDC) is also implemented within the dataflow architecture.

Data-Processing Unit: This module implements real-time data processing tasks, such as artificial intelligence engines or any high data processing acceleration. It has an AXI4-Stream interface with the Data-Flow module and an AXI4-Lite interface to communicate with the PS. For instance, in the open-interface CT use case, we have used it to implement the image pre-processing step described in our previous work in [37].

B. SOFTWARE ARCHITECTURE

This section describes the software architecture running on the CCU. The software architecture consists of the *OS layer* and the *Application layer*.

The **OS layer** supports one or more OSs and bare-metal applications running on CPU cores in the PS. In addition, if the selected MPSoC-FPGA supports hypervisors, it is deployed at this layer. This layer also contains the First System Boot Loader (FSBL), which is responsible for configuring and booting the PL part and PS parts of the MPSoC-FPGA. The OS layer provides the scheduler for the different tasks and communication modules. For instance, the user sets up a structure (e.g., C struct) that contains all the communication and scheduler parameters for the *Software layer*. The communication parameters consist of the static or dynamic connection (e.g. DHCP, IP, TCP port per slave-client node) implemented through an input configuration file. Furthermore, the type of scheduler and default priorities for each task are also set, as well as possible defaults for other settings.

The **Application layer** implements the server for the *client-server class* and the proposed datagram protocol for the *Application non-real-time class*. In fact, it executes the non-real-time tasks that mainly interact with slave-client nodes. In order to manage the different tasks associated with the different slave-client nodes in our software, we have divided the *Application layer* of the CCU architecture into four modules: *Communication*, *Timer*, *Command*, and *Execute*. Each module is responsible for a specific part of the communication as described below:

- **Communication module**: This module is responsible for creating server-socket instances and accepting all incoming client connections.
- **Command module**: Within the server-socket instance, it handles the communication with the connected client, which can run a single command thread at that time. For this reason, there is one "*singleton instance*" [38] per client. All received messages are decoded and passed to the specific *Execute* thread by using the FSM shown in Fig. 11.
- **Execute module**: This module executes the operations related to the received command task. These operations realize the "*business logic layer*" [39] of the CPS software architecture. For this purpose, non-real-time tasks are mapped on the PS, and real-time tasks on the PL. Furthermore, this module is also responsible for

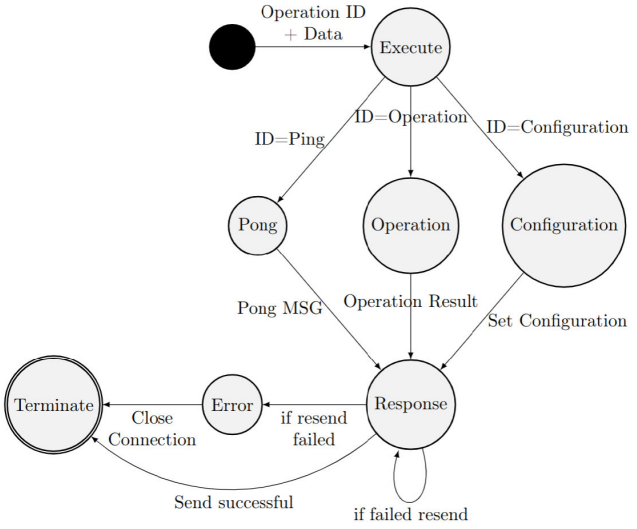

FIGURE 11. States of the *Execute* module during the execution of a received and decoded message from a slave-client node.

controlling and interacting with the hardware modules implemented on the PL part and accessed via the AXI4-Lite interface. Finally, it sends the response message of the requested command task, the operation status, and the requested data to the client. To manage all these execution steps, we have implemented the FSM shown in Fig. 11 that uses different states for each execution step.

- **Timer module**: During the communication, this module manages `TIME_OUT` events. It triggers the timer task that is set to N seconds, and it resets the timer `COUNTER` for each correct message or Ping message received from the respective client. When a `TIME_OUT` event occurs, the client connection is closed, and the tasks of the corresponding *Execute* and *Command* modules are terminated.

These software modules run on a Real-time Operating System (RTOS), using different instances and TCP ports for each slave-client node. In this way, different instances can be associated with different isolated domains, and the software architecture is scalable by supporting priority handling of run-time tasks. For managing the different tasks and priorities at run-time, the RTOS uses a task scheduler that is set up before the system execution starts. When the system is running, each module sets the following priority hierarchy:

1. *Timer module* (`Real-Time priority`)
2. *Command module* (`High priority`)
4. *Execute module* (`Medium Priority`)
5. *Communication module* (`Low Priority`)

This hierarchy prevents software starvation caused by running operations that depend on unresponsive input commands. Furthermore, to avoid a deadlock race condition caused by a `TIME_OUT` event, the timer generates an interrupt that terminates all operations of the other modules for the lost connection. It uses the RTOS timer to do this. In addition, when an *Execute* module is running, a priority schema between commands is set based on the operation ID of the received datagram message.

C. EXECUTION EXAMPLE OF THE CENTRAL CONTROL UNIT

A typical scenario involving all CCU layers and several slave-client nodes is shown in Fig. 12. In this UML sequence diagram, we have two slave-client nodes: the first (left side) exchanges non-real-time messages through the Application Datagram Protocol (client-server protocol), and the second (right side) exchanges real-time control/synchronization messages using the Handshake Protocol (master-slave protocol). It reports a case where real-time and non-real time tasks interact without affecting deadlines and using software and hardware modules. In Fig. 12, the colors of the black, orange, blue, and green arrows represent the powering-up process of the CPS, the client-server communication, the communication of internal software and hardware tasks with associated threads, and the master-slave communication between the CCU and a slave-client node, respectively.

At the moment that the CPS is powered up, the FSBL inside the CCU configures the PL part and boots the PS part with the RTOS. After all internal hardware/software modules are configured, they send the setup configuration to all the slave-client nodes, as shown by the black arrow in Fig. 12. In the UML sequence diagram, we have shown only two nodes with separated interfaces for ease of reading.

Once all the nodes are ready, the *Communication module* creates a server socket for each client connection, and it waits for N other client connections according to the configuration file. For each client connection established and server socket created, the *Communication module* creates a specific client socket that starts the *Command module* thread. The *Command module* also creates a specific client timer task that waits for messages from the newly accepted client.

When a message arrives, the *Command module* decodes the message and checks the CRC packet, according to the datagram protocol explained in Sec. III-D1. In case of a CRC error, the CCU discards the message and sends another error message to the client. For each decoded command, the operation ID and data are passed by the FSM placed in the *Execute module*, as shown in Fig. 11; to do so, a thread with the received command data is created. Furthermore, each created thread is stored in an array for all running operations using the `Execute module thread` to handle multiple operations. This module performs the predefined operation associated with the operation ID on the internal and external hardware and stores received data. When the operation is finished, a response message with the stored data and success status is sent to the client based on the datagram protocol. After successful transmission, the *Execute module* deletes all stored data, and the thread is terminated and removed from the array of running operations.

If the response message transmission fails, two additional attempts are made with a delay of 500ms. If both additional attempts fail, the *Execute module thread* triggers the

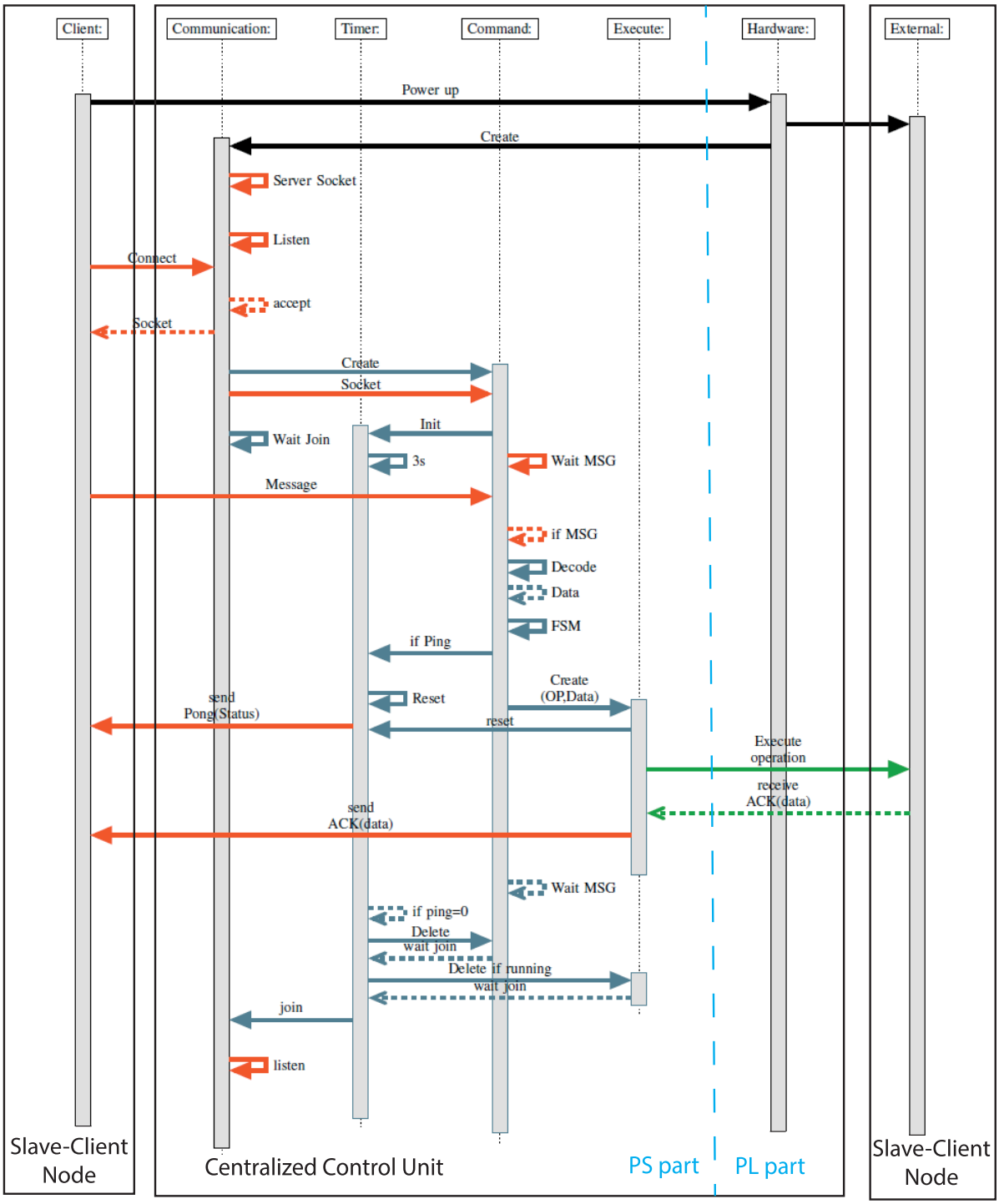

FIGURE 12. UML sequence diagram for the client-server communication and the interaction between modules in operation. Orange indicates TCP/IP operations, green PL and external hardware communications, and blue PS parts.

`TIME_OUT` software interrupt and then terminates all the related threads.

V. CASE STUDY: OPEN-INTERFACE CT

This section illustrates our Communication Infrastructure and the CCU in an open-interface CT scanner assembled in our laboratory.

The aim of having an open-interface CT is to provide a CT scanner that supports real-time control and data processing for exploring new clinical acquisition techniques such as medical intervention and multi-modality imaging [20], [40]. In addition, doctors or researchers can use internal parameters that are not accessible in commercial CT scanners. In fact, our Communication Infrastructure and CCU enable real-time

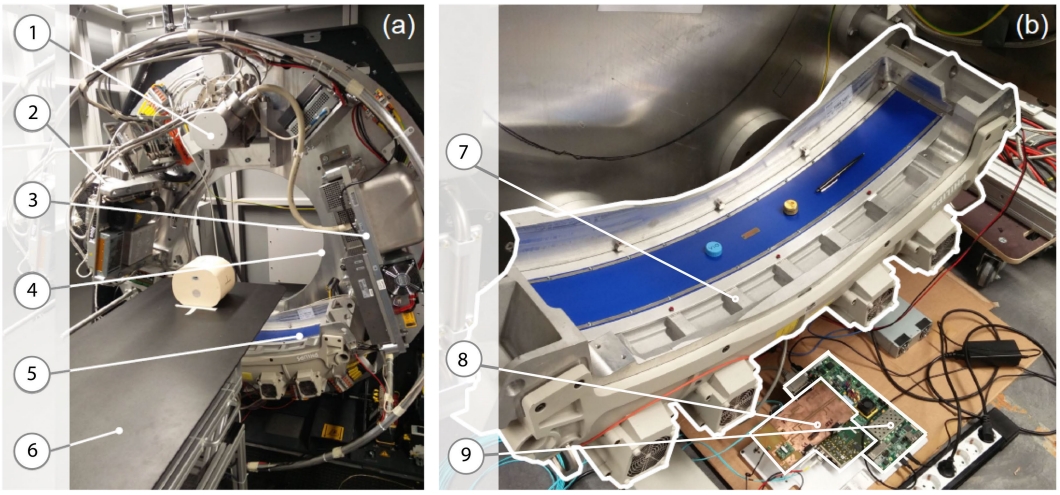

FIGURE 13. Components of our open-interface CT system. (a): Experimental CT complete system with (1) X-ray tube, (2) Cooling system, (3) Generator, (4) Gantry subsystem, (5) Multiline DMS, (6) Patient table. (b): Detailed view of the DMS and CCU implemented on the Xilinx ZC706 Evaluation Kit: (7) Multiline DMS, (8) FPGA Extension-board, (9) Xilinx ZC706 Evaluation Kit.

capabilities and allow designers and researchers to integrate new components into the existing open-interface CT as plug-and-play sub-modules. For this purpose, the discussion will also analyze the cost of adding an additional flat-panel detector to our open-interface CT scanner.

The CT scanner is a CPS mainly composed of the following subsystem components: patient table, gantry subsystem, DMS, X-ray tube subsystem, collimator, and reconstruction subsystem, as shown in Fig. 13. The gantry module, patient table, and the image reconstruction system are fixed on the ground, which is called stationary side. All other components are mounted on the rotating disk of the gantry, which is called rotating side [41], [42]. These two sides communicate with each other using the slip-ring technology [43], [44], which is an electromechanical device capable of transmitting high-speed communication signals. It consists of concentric circular rings parallel to the gantry axis [42].

A. COMMUNICATION INFRASTRUCTURE

In the open-interface CT, the components are modeled as independent interconnected nodes. To properly interconnect them according to their interfaces and protocols, we have utilized the proposed Communication Infrastructure, described in Sec. III. Fig. 14 shows the diagram with the nodes, their interfaces and protocols, and their interconnection links. The non-real-time interfaces use Ethernet, which requires a switch to interconnect them.

As described above, the communication between the stationary and rotating sides is done via slip-ring technology. Due to the fact that is expensive and is limited in speed and number of transceivers, it is the communication bottleneck of the CT. In addition, all the CT sensors and actuators, such as the DMS and X-ray tube, which need to be controlled and synchronized in real-time, are located on the

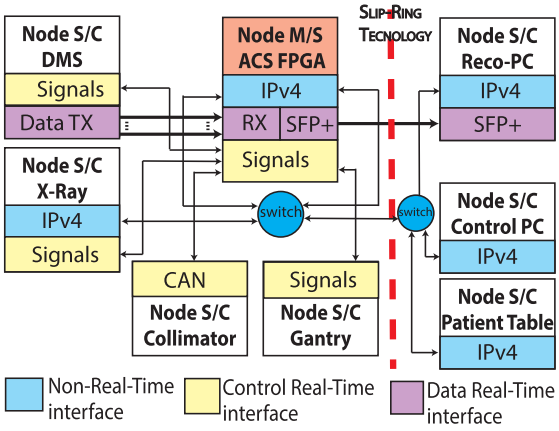

FIGURE 14. Node interconnection and interface layer for the open-interface CT.

rotating side of the gantry. For these reasons, we placed our master-server node (implemented by the CCU) of the Communication Infrastructure on the rotating side. As shown in Fig. 14, the CCU communicates with the stationary side using the Application Datagram Protocol and the non-real-time interface realized over the Gigabit Ethernet.

Furthermore, the CCU collects image data from the DMS node and forwards them to the Reconstruction Unit node, which we have designed as a slave-client node in our Communication Infrastructure, called Reco-PC in Fig. 14. For this communication, we use the Application Stream Protocol at the application layer, while at the Interface layer, we have implemented a Gigabit Transceiver (GTX) [45] in the PL part of the MPSoc-FPGA. In this specific case, it streams data at a rate of 6.250 Gbit/sec with 8b/10b encoding using a Small form-factor pluggable (SFP+) port.

During the image acquisition, to control and synchronize the DMS, the collimator, and the X-ray tube with the

gantry position, we use the Control master-slave class over the Real-time control interface class, where messages are sent as synchronization signals. All these components, as shown in Fig. 14, have a custom physical interface with a custom protocol provided by the vendor and implemented in the CCU with the enhancement of the handshake. For implementing the Handshake Protocol, we have utilized two independent FSMs: the first is responsible for the retransmission mechanism, and the second is for the control flow transmission.

B. CENTRALIZE CONTROL UNIT

The CCU manages all tasks and interactions within our open-interface CT. As shown in Fig. 13, the CCU runs on the Xilinx Evaluation board ZC706 having the FPGA model XC7Z045. In the *Hardware layer*, the Control/Synchronization Unit and the Data-Flow Unit are modeled at Register-Transfer Level (RTL) and described using SystemVerilog. Instead, the Data Processing Unit is described with C and implemented using High-Level Synthesis (HLS). The OS and Application layers are implemented on the Application Processing Unit, which has a dual-core Cortex-A9. Since the selected MPSoC-FPGA has no isolation mechanism, we have used Protection Units (PUs) IP cores, as proposed in our previous work [46]. These PUs provide isolation support between PS-PL and PL-PL communication for MPSoC-FPGAs of the AMD-Xilinx series 7 family. Fig. 15 shows the proposed CCU architecture for the open-interface CT, where all the module instances for each CCU layer of the open-interface CT have been reported.

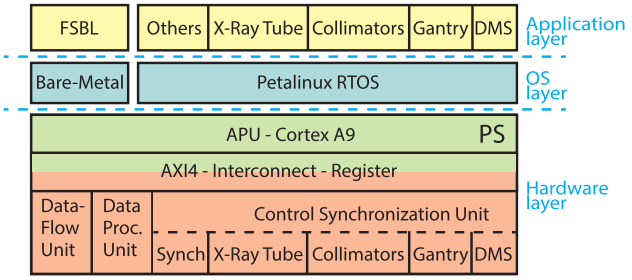

FIGURE 15. Illustration of the CCU architecture for the open-interface CT.

In the *Hardware layer*, the Control-Synchronization Unit has isolated modules for each slave-client node that communicates with the master-slave paradigm. The only shared module is the synchronization module, which handles the enabling signals for starting and stopping the X-ray tube during acquisition and for synchronizing the gantry encoder, collimator, X-ray tube, and DMS. In addition, if an error occurs during the acquisition, this module is responsible for catching it and stopping the acquisition in the next clock cycle. In relation to the Communication Infrastructure, this Unit realizes the Real-time control interface class and the Control master-slave class. The DMS, collimator, gantry encoder, and X-ray tube use custom interfaces and protocols associated with the Real-time control interface class and the Control master-slave class. For the custom interfaces, we have built an extension board, shown in Fig. 13, which is connected through an FPGA Mezzanine Card (FMC+) Interface using the VITA 57.4 FMC+ Standard. In this way, we have connected the custom interfaces to the FPGA GPIO ports and implemented the transceivers inside each related hardware module. These transceivers are also modeled at RTL. The *Hardware layer* also implements the Data-Flow Unit, where the data are collected from the DMS and forwarded to the Data-Processing Unit and the External Reconstruction Unit. In the actual implementation, the Data-Processing Unit implements the I0-correction algorithm [37], which is a real-time data processing task.

On the *OS layer*, we tested Petalinux RTOS and FreeRTOS. FreeRTOS provides a powerful and small kernel without any additional layers provided by the OS. In fact, under FreeRTOS, tasks written as bare-metal source code access the physical memory address space directly, while Petalinux provides a virtual memory address space. On the other hand, Petalinux provides a small Unix kernel with high support of libraries running in C/C++. In terms of timing performance, both RTOSs gave us the same results, but Petalinux offered us compatibility with external libraries provided by the X-ray tube manufacturer and a simpler environment for future implementations using additional sensors/actuators. Between these RTOSs, we chose Petalinux for its compatibility and scalability features which are important requirements for the open-interface CT and multi-modality imaging research.

On the *Application layer*, we have implemented the non-real-time tasks associated with the real-time communication classes. As shown in Fig 15, there is a software module for each sub-module of the Control Synchronization Unit. Furthermore, the *Application layer* implements the server and the Datagram protocol for the *Application non-real-time class*. The server uses the configuration file explained in Sec. IV-B, where we set static IP addresses for all the nodes. In the server, for decoding the messages of the communication tasks, we configure a *struct* that describes the acceptable data packet. The Data packets used for the open-interface CT are shown in Fig. 16.

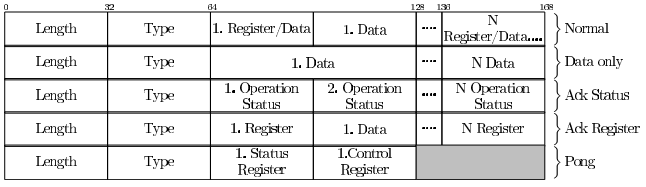

FIGURE 16. Structure of four typical message data sections and Pong with exemplary bit-sizes used in the open-interface CT.

In the Communication module, the maximum number of client connections is set to 4, which is the number of clients in our system architecture in Fig. 14. Separating the connections into different sockets gives us an independent endpoint for each major acquisition system control. This prevents unexpected connections from being opened, and if malicious connections try to connect to the server, they will be

rejected. In addition, if multiple connections attempt to access the same socket and the number of connections exceeds the maximum, then the open-interface CT is put into safe mode to prevent unnoticed CT operations.

Due to the safety requirements of the X-ray tube and the rest of the system, we have set the *TIME_OUT* of the Timer module to 2 seconds. In case of a *TIME_OUT* event, the CT is also set in safe mode. For doing this, the DMS and the X-Ray Tube are disabled, and the CCU hardware and software modules are set to default values and/or are deactivated.

VI. ANALYSIS AND DISCUSSION

In this section, we discuss the evaluation for the realization of our Communication Infrastructure and the CCU for the open-interface CT, and we compare them in terms of qualitative analysis with the proposed solution of AMD-Xilinx [29]. Finally, to evaluate the plug-and-play feature of our Communication Infrastructure and the CCU, we analyze the effort of adding another detector into the open-interface CT.

AMD-Xilinx in [29] proposes a system architecture for CT, based on a distributed control unit. As shown in Fig. 17, the CT scanner consists of 4 subsystem components: *High Voltage (HV) Supply Control*, *Data Acquisition & Gantry Control*, *Image Reconstruction* and *System Sequencer*. The Contro units is distributed on three of these components.

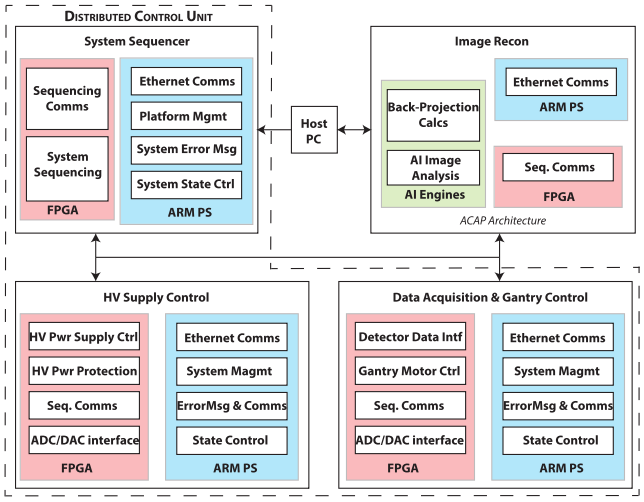

FIGURE 17. AMD-Xilinx system architecture for CT [29].

The *HV Supply Control* manages all the tasks inside the X-ray tube: software parameters, high voltage supply, and errors. For these tasks, they propose to use an MPSoC-FPGA that can manage the HV modules in the PL and the software tasks (e.g., error messages, set-up parameters, communication management) in the PS part. The *Data Acquisition & Gantry Control* controls the DMS, collects the data from it, and forwards them to the reconstruction system. The *Image Reconstruction* and the *System Sequencer* implement the reconstruction algorithm and the synchronization tasks within the CT.

In contrast to the AMD-Xilinx architectural model, we have implemented all data acquisition and control tasks in the CCU. The proposed architecture handles time issues of the various tasks, proposing different classes for real-time control and data tasks and non-real-time tasks. This enables us to meet real-time requirements and facilitates the integration of additional components. For safety issues, the various units responsible for different components are isolated through the Protection Unit proposed by authors in [46]. In fact, with the proposed solution, the slave-client nodes, such as the DMS, X-ray tube, and the HV module, must only implement the FSMs required for controlling the internal modules and setting them for the safe mode. In our system architecture, like AMD-Xilinx, we have a reconstruction unit for reconstructing the image, but it is a slave-client node that does not participate in the synchronization.

If an additional component (e.g., DMS or X-ray tube) needs to be integrated into the AMD-Xilinx CT system architecture, all subsystem components must be modified. Furthermore, new interfaces must be placed on all these components according to the vendor protocols of the new component. Conversely, with our CCU hardware/software architecture and the proposed Communication Infrastructure, we can easily add components in the open-interface CT as plug-and-play modules and nodes, respectively.

In addition, the proposed communication infrastructure can be implemented with custom and standardized protocols and there are no architecture assumptions on the protocol like the OPC UA, which is limited to CPS where all vendor components communicate with their client-server protocol. In our communication infrastructure, we can integrate OPC UA at the application layer, but we can also use components, that do not have any client-server interface, such as the DMS in the open-interface CT.

Furthermore, with the proposed structure of the nodes and application protocols, the CCU can guarantee the required bandwidth to all the different nodes and types of tasks, e.g., real-time and non-real-time tasks, control/synchronization, and data tasks. In addition, the proposed protocols enhance the security of the CPS at the application protocol layer. The datagram protocol provides a CRC at the application layer where TCP/IPv4 security mechanisms are insufficient. For example, if a malicious node tries to send an unexpected command to the CCU, the TCP/IPv4 does not implement a security feature to block it, but the Application Datagram Protocol and the server on the CCU can block it. In fact, they check the CRC, the ID Version of the command, the port that must be different for each node, and the number of nodes that are fixed. The Handshake Protocol also defines a common handshake mechanism for master-slave communication by guaranteeing that messages from the master to the slave are correctly sent and received. If a message is lost or corrupted at the link level, the Handshake Protocol triggers the retransmission or notifies the error to the CCU *Control-Synchronization Unit*.

In addition to the qualitative analysis, we also consider a quantitative analysis for the implementation of our CCU architecture on the Xilinx ZC706 board, including the FPGA model XC7Z045 [47]. For the synthesis, implementation, and quantitative analysis, we have used the Vivado tool [48] from AMD-Xilinx. To meet the timing requirements of the various external components and the processing unit given by the open-interface CT, we use different clock frequencies for the different hardware units. The Control-Synchronization Unit uses a reference clock of 100 MHz. Within this unit, different clock lines are implemented for the different transceivers of the DMS, the X-ray tube, the gantry, and the collimators. The Data Processing Unit uses a clock frequency of 200 MHz. The Data-Flow Unit uses different clock lines that are required for the various transceivers associated with the external components and the CDC: the user clock frequency is 100 MHz, and the reference clock frequency for the transceivers is set to 156.250 MHz and 78.150 MHz. By setting these clock frequencies and with the support of the CDC, the data are collected and processed on-the-fly, meeting the real-time constraints.

In addition, the implementation of the CCU architecture uses a low percentage of resource utilization for the PL part. As shown in Fig. 18, the proposed CCU uses less than 10% of resources for most types of FPGA slices. The resource utilization doesn't take into account the Data Processing Unit because it only depends on the selected CPS application and does not affect the plug-and-play capability. The low resource utilization is the result of optimizing the mapping of real-time and non-real-time tasks executed on PL and PS parts, respectively. This result is essential for the plug-and-play feature of the CPS, as it facilitates the scalability of the systems when new sensors/actuators are added.

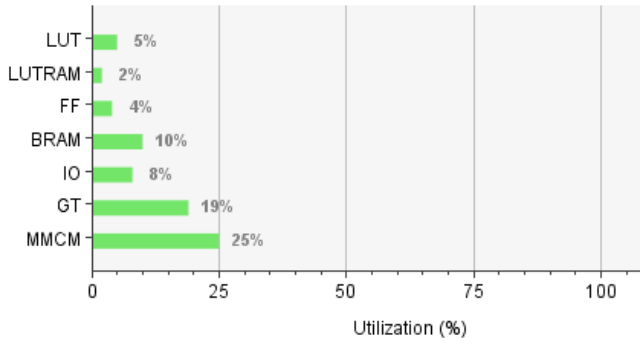

FIGURE 18. Resources' utilization for the different slices in the CCU implementation on the FPGA models XC7Z045 [47] (Vivado report).

In our CCU, we didn't consider the Image Processing Unit presented in [37] because it is not involved in CPS communication, and it is not in the scope of this work. Reference [37] also describes the configuration for using the open-interface CT scanner, where different phantom data are acquired and pre-processed with the proposed CCU, and evaluated in terms of image quality. Moreover, in other CPS applications, the available resources can be used for implementing any real-time data processing tasks involved.

A. SYSTEM EXTENSION

To evaluate the proposed Communication Infrastructure and the CCU in terms of plug-and-play capabilities, we consider the effort of adding an additional DMS in the assembled Open-Interface CT. In addition, we make some considerations of adding the same DMS in a CT scanner that uses the AMD-Xilinx CT architectural model, shown in Fig. 17.

For this study, we have selected the XC-Thor photon counting detector [49]. It uses a Gigabit Ethernet interface for non-real-time and real-time data communication and a custom signal interface for real-time control/synchronization communication.

In order to integrate this DMS in the open-interface CT, we started adding it to the proposed Communication Infrastructure. Here, the DMS represents a new slave-client node having vendor interfaces and protocols at the interface and communication layers. To differentiate the different interfaces and communication tasks, we associate them with the proposed classes for each node. Finally, on top of them we use the Handshake Protocol, the Application Stream Protocol, and the Datagram Protocol.

To connect the new node to the system, we connect the Gigabit Ethernet interface to the switch that is on the rotating side. The custom control/synchronization interface is connected directly to the GPIO pins of the ZC706 Evaluation Board. The GPIO pins are mapped as PL input/output ports. In this way, the vendor protocols for the different interfaces can be instantiated as IP cores in the PL part. The transceiver for the Real-Time control interface class is instantiated in the Control Synchronization Unit of the CCU, where the Handshake Protocol can handle errors and lost messages. The Application non-real-time class and the Data Real-Time Class access the Gigabit Ethernet interface concurrently. In setup mode, the non-real-time tasks communicate over the Application Datagram Protocol, while the Application Real-Time Data Class and its Stream Protocol are used during the Data Acquisition.

In addition to the effort of adding the instances for the different interfaces and protocols in the CCU for the interface layer and the transport protocol layer, these have to be mapped to the PS part as IO memory-mapped devices. Furthermore, the software layer of the CCU architecture also should be extended with an additional module for the new DMS. This module contains the control logic for the new detector, which is not discussed here, and the communication part for the server.

Initiating the new module, a configuration file is defined for the DMS. The file contains the connection settings, thread scheduling, and necessary data packets related to the vendor and system requirements. In this way, the server module is set up to run independently of the first DMS on a single thread of the CCU. To enable communication with the PL and PS parts

of the extension, the existing FSBL must be extended with its configuration to initiate both parts. Within the software layer, the four modules of the server are adjusted to the new DMS, fulfilling the vendor requirements.

The communication with the control side is established via a new Ethernet port and a maximum number of clients by the configuration file in the communication module.

The commands of the new DMS are translated into the structure of the Execute Module and linked to the CCU PL based on the previous FSBL setup. In addition to the Execute Module, the operation ID for the communication is added to the command datagram, and/or new custom datagram structures are set with the decode/encode functions from the configuration file. Similarly, the operation IDs are mapped to the command of the Execute Module, and schedule options are configured.

In the final step, the timer module is set with vendors' timeout in the hardware timer, while the timer for the communication timeout remains the same. While in our proposed work, all the hardware/software extensions affect only the CCU, in the AMD-Xilinx architectural model, all components are involved because HV supply control, sequencer, data acquisition, and gantry control tasks are implemented on different physical components. As a result, the integration effort and costs are higher, and it is not possible to provide the plug-and-play feature.

VII. SUMMARY

This work presented a Communication Infrastructure and a CCU hardware/software architecture for MPSoC-FPGA, providing a solution for the plug-and-play capability in CPSs. It supports a wide range of CPS applications that have numerous components with different interfaces and custom protocols. Moreover, for the Communication Infrastructure, various application protocols are proposed for control/synchronization real-time, data real-time, and non-real-time tasks. It also shows how to implement the various proposed protocols and how to separate the different tasks inside and between PS and PL for the CCU hardware/software architecture. For the evaluation scope, the use case of the open-interface CT, assembled in our laboratory, has been considered. Due to the fact that tasks are mapped between PS and PL, the PL utilizes less than 10% of Flip-Flops and LookUp-Tables in the PL. The low resource's utilization allows adding components and image processing tasks to explore new clinical acquisition techniques such as multi-modality imaging. Finally, this work discusses and evaluates how a CPS component, such as a flat panel detector, can be added to the open-interface CT in a "plug-and-play" fashion. The proposed Communication Infrastructure and the CCU architecture define the base for a design framework in CPS applications, where custom and standard protocols coexist, and where the proposed protocols can be utilized for integrating them. In fact, designers can use it for other CPS applications.

REFERENCES

[1] P. Cicconi, A. C. Russo, M. Germani, M. Prist, E. Pallotta, and A. Monteriu, "Cyber-physical system integration for industry 4.0: Modelling and simulation of an induction heating process for aluminium-steel molds in footwear soles manufacturing," in *Proc. IEEE 3rd Int. Forum Res. Technol. Soc. Ind. (RTSI)*, Sep. 2017, pp. 1–6.

[2] F. Hofer, "Architecture, technologies and challenges for cyber-physical systems in industry 4.0: A systematic mapping study," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, M. Oivo, D. Méndez, and A. Mockus, Eds., New York, NY, USA, 2018, pp. 1–10.

[3] J. Jamaludin and J. M. Rohani, "Cyber-physical system (CPS): State of the art," in *Proc. Int. Conf. Comput., Electron. Electr. Eng. (ICE Cube)*, Nov. 2018, pp. 1–5.

[4] F. Hu, Y. Lu, A. V. Vasilakos, Q. Hao, R. Ma, Y. Patil, T. Zhang, J. Lu, X. Li, and N. N. Xiong, "Robust cyber–physical systems: Concept, models, and implementation," *Future Gener. Comput. Syst.*, vol. 56, pp. 449–475, Mar. 2016.

[5] N. Dey, A. S. Ashour, F. Shi, S. J. Fong, and J. M. R. S. Tavares, "Medical cyber-physical systems: A survey," *J. Med. Syst.*, vol. 42, no. 4, pp. 1–13, Mar. 2018.

[6] D. Brasse, B. Humbert, C. Mathelin, M.-C. Rio, and J.-L. Guyonnet, "Towards an inline reconstruction architecture for micro-CT systems," *Phys. Med. Biol.*, vol. 50, no. 24, pp. 5799–5811, Dec. 2005.

[7] R. Baheti and H. Gill, "Cyber-physical systems," *Impact Control Technol.*, vol. 12, no. 1, pp. 161–166, 2011.

[8] S. Ali, T. A. Balushi, Z. Nadir, and O. K. Hussain, "Distributed control systems security for CPS," in *Cyber Security for Cyber Physical Systems* (Studies in Computational Intelligence). Cham, Switzerland: Springer, 2018, pp. 141–160.

[9] S.-H. Tseng and J. Anderson, "Synthesis to deployment: Cyber-physical control architectures," 2020, *arXiv:2012.05211*.

[10] S. S. Jogwar and P. Daoutidis, "Community-based synthesis of distributed control architectures for integrated process networks," *Chem. Eng. Sci.*, vol. 172, pp. 434–443, Nov. 2017.

[11] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration*, vol. 59, pp. 148–156, Sep. 2017.

[12] E. R. Griffor, C. Greer, D. A. Wollman, and M. J. Burns, "Framework for cyber-physical systems: Volume 1, overview," Nat. Inst. Standards Technol., U.S. Dept. Commerce, Gaithersburg, MD, USA, Tech. Rep. 1500-201, 2017.

[13] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Berlin, Germany: Springer, 2009.

[14] M. Hoffmann, S. Malakuti, S. Grüner, S. Finster, J. Gebhardt, R. Tan, T. Schindler, and T. Gamer, "Developing industrial CPS: A multi-disciplinary challenge," *Sensors*, vol. 21, no. 6, p. 1991, Mar. 2021.

[15] B. Bordel, D. S. De Rivera, and R. Alcarria, "Plug-and-play transducers in cyber-physical systems for device-driven applications," in *Proc. 10th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput. (IMIS)*, Jul. 2016, pp. 316–321.

[16] E. Armengaud, G. Macher, A. Massoner, S. Frager, R. Adler, D. Schneider, S. Longo, M. Melis, R. Groppo, F. Villa, P. O'Leary, K. Bambury, A. Finnegan, M. Zeller, K. Höfig, Y. Papadopoulos, R. Hawkins, and T. Kelly, "DEIS: Dependability engineering innovation for industrial CPS," in *Advanced Microsystems for Automotive Applications*. Cham, Switzerland: Springer, 2018, pp. 151–163.

[17] *Medical Device 'Plug-and-Play (MD PNP) Interoperability Program*. Accessed: Jan. 17, 2022. [Online]. Available: https://mdpnp.org/team.html

[18] R. M. Hofmann, "Modeling medical devices for plug-and-play interoperability," Ph.D. thesis, Massachusetts Inst. Technol., Cambridge, MA, USA, 2007.

[19] T. Li, F. Tan, Q. Wang, L. Bu, J.-N. Cao, and X. Liu, "From offline toward real-time: A hybrid systems model checking and CPS co-design approach for medical device plug-and-play (MDPnP)," in *Proc. IEEE/ACM 3rd Int. Conf. Cyber-Phys. Syst.*, Apr. 2012, pp. 13–22.

[20] G. Wang, M. Kalra, V. Murugan, Y. Xi, L. Gjesteby, M. Getzin, Q. Yang, W. Cong, and M. Vannier, "Vision 20/20: Simultaneous CT-MRI—Next chapter of multimodality imaging," *Med. Phys.*, vol. 42, no. 10, pp. 5879–5889, Oct. 2015.

[21] GE Healthcare. *How Clinicians See Today's CT Challenges-Article*. Accessed: Jan. 5, 2023. [Online]. Available: https://www.gehealthcare.ae/insights/article/how-clinicians-see-today%E2%80%99s-ct-challenges
[22] E. Alcaín, P. R. Fernández, R. Nieto, A. S. Montemayor, J. Vilas, A. Galiana-Bordera, P. M. Martinez-Girones, C. Prieto-de-la Lastra, B. Rodriguez-Vila, M. Bonet, C. Rodriguez-Sanchez, I. Yahyaoui, N. Malpica, S. Borromeo, F. Machado, and A. Torrado-Carvajal, "Hardware architectures for real-time medical imaging," *Electronics*, vol. 10, no. 24, p. 3118, 2021.
[23] R. Gupta, C. Walsh, I. S. Wang, M. Kachelrieß, J. Kuntz, and S. Bartling, "CT-guided interventions: Current practice and future directions," in *Intraoperative Imaging and Image-Guided Therapy*. New York, NY, USA: Springer, 2014, pp. 173–191.
[24] M. V. García, E. Irisarri, F. Pérez, E. Estévez, and M. Marcos, "OPC-UA communications integration using a CPPS architecture," in *Proc. IEEE Ecuador Tech. Chapters Meeting (ETCM)*, Oct. 2016, pp. 1–6.
[25] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for Tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, Mar. 2020.
[26] R. V. Chakaravarthy, H. Kwon, and H. Jiang, "Vision control unit in fully self driving vehicles using Xilinx MPSoC and opensource stack," in *Proc. 26th Asia South Pacific Design Autom. Conf. (ASPDAC)*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 311–317.
[27] Y. Liu, Y. Peng, B. Wang, S. Yao, and Z. Liu, "Review on cyber-physical systems," *IEEE/CAA J. Autom. Sinica*, vol. 4, no. 1, pp. 27–40, Jan. 2017.
[28] *Zynq UltraScale+MPSoC Data Sheet: Overview, DS891, Rev. 1.9*, AMD-Xilinx, Xilinx, Inc., San Jose, CA, USA, May 2021.
[29] *Medical Imaging With CT Scanners and MRI Machines*. Accessed: Mar. 28, 2021. [Online]. Available: https://www.xilinx.com/applications/medical/medical-imaging-ct-mri-pet.html
[30] A. Gatouillat, Y. Badr, B. Massot, and E. Sejdic, "Internet of Medical Things: A review of recent contributions dealing with cyber-physical systems in medicine," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 3810–3822, Oct. 2018.
[31] E. Min, K. Kim, H. Lee, H.-I. Kim, Y. H. Chung, Y. Kim, J. Joung, K. M. Kim, S.-K. Joo, and K. Lee, "Development of compact, cost-effective, FPGA-based data acquisition system for the iPET system," *J. Med. Biol. Eng.*, vol. 37, no. 6, pp. 858–866, Jun. 2017.
[32] G. Korcyl et al., "Trigger-less and reconfigurable data acquisition system for positron emission tomography," *Bio-Algorithms Med-Syst.*, vol. 10, pp. 37–40, Jan. 2014.
[33] E. Fysikopoulos, G. Loudos, M. Georgiou, S. David, and G. Matsopoulos, "A spartan 6 FPGA-based data acquisition system for dedicated imagers in nuclear medicine," *Meas. Sci. Technol.*, vol. 23, no. 12, Nov. 2012, Art. no. 125403.
[34] M. M. Alani, "TCP/IP model," in *Guide to OSI and TCP/IP Models*. Cham, Switzerland: Springer, 2014, pp. 19–50.
[35] *Amba 4 Axi4-Stream Protocol, Version: 1.0 Specification*, ARM Ltd., Cambridge, U.K., 2010.
[36] D. Passaretti and T. Pionteck, "Configurable pipelined datapath for data acquisition in interventional computed tomography," in *Proc. IEEE 29th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2021, p. 257.
[37] D. Passaretti, M. Ghosh, S. Abdurahman, M. L. Egito, and T. Pionteck, "Hardware optimizations of the X-ray pre-processing for interventional computed tomography using the FPGA," *Appl. Sci.*, vol. 12, no. 11, p. 5659, Jun. 2022.
[38] K. Stencel and P. Węgrzynowicz, "Implementation variants of the Singleton design pattern," in *Proc. OTM Confederated Int. Conf. Move Meaningful Internet Syst.* Berlin, Germany: Springer, 2008, pp. 396–406.
[39] S. T. Albin, *The Art of Software Architecture: Design Methods and Techniques*, vol. 9. Hoboken, NJ, USA: Wiley, 2003.
[40] D. Passaretti and T. Pionteck, "A control data acquisition system architecture for MPSoC-FPGAs in computed tomography," in *Proc. Int. Symp. Appl. Reconfigurable Comput.* Cham, Switzerland: Springer, 2023, pp. 361–365.
[41] J. R. Wesolowski and M. H. Lev, "CT: History, technology, and clinical aspects," in *Seminars in Ultrasound, CT and MRI*, vol. 26. Amsterdam, The Netherlands: Elsevier, 2005, pp. 376–379.
[42] D. Passaretti, J. M. Joseph, and T. Pionteck, "Survey on FPGAs in medical radiology applications: Challenges, architectures and programming models," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 279–282.
[43] L. Faggioni, F. Paolicchi, and E. Neri, *Elementi di Tomografia Computerizzata*, vol. 4. Milan, Italy: Springer, 2011.
[44] Schleifring. *Schleifring CT Gantry*. Accessed: Oct. 1, 2023. [Online]. Available: https://www.schleifring.de/fileadmin/08_Downloads/CT-Applications_January18.pdf
[45] *Z7 Series FPGAs GTX/GTH Transceivers User Guide (UG476)*, AMD-Xilinx, Xilinx, Inc., San Jose, CA, USA, Aug. 2018.
[46] D. Passaretti, F. Böhm, M. Wilhelm, and T. Pionteck, "Hardware isolation support for low-cost SoC-FPGAs," in *Proc. 35th Int. Conf. Archit. Comput. Syst. (ARCS)*, Heilbronn, Germany. Cham, Switzerland: Springer, Sep. 2022, pp. 148–163.
[47] "ZC706 evaluation board for the Zynq-7000 XC7Z045 SoC," Xilinx, Inc., San Jose, CA, USA, Tech. Rep. UG954 (v1.8), Aug. 2019.
[48] T. Feist, "Vivado design suite," Xilinx Inc., San Jose, CA, USA, White Paper WP416(v1.1), 2012, p. 30, vol. 5.
[49] DCA Varex Image Company. *XC-Thor Photon Counting X-Ray Detector*. Accessed: Nov. 16, 2022. [Online]. Available: https://www.vareximaging.com/wp-content/uploads/2022/01/XC-THOR_PDS.pdf

DANIELE PASSARETTI (Graduate Student Member, IEEE) received the M.S. degree (cum laude) in computer engineering from the University of Naples Federico II, Italy, in 2017. He is currently pursuing the Ph.D. degree with Otto-von-Guericke University Magdeburg, Germany. He is also a Research Assistant with Otto-von-Guericke University Magdeburg. His research interests include digital design, medical device design, computer architecture, and reconfigurable technologies.

MAX STEIGER was born in Lübeck, Germany, in 1995. He received the B.S. and M.S. degrees in medical system engineering from Otto-von-Guericke University Magdeburg, Germany, in 2019 and 20222, respectively. He is currently working on digitalizing medical care with the University Hospital Magdeburg, Germany. Since 2023, he has been a Research Assistant with the Faculty of Computer Science, Otto-von-Guericke University, as a sideline, focusing on medical image processing, and tracking in interventional CT.

THILO PIONTECK (Member, IEEE) received the Diploma and Ph.D. (Dr.-Ing.) degrees in electrical engineering from Technische Universität Darmstadt, Germany, in 1999 and 2005, respectively. In 2008, he was appointed as an Assistant Professor in integrated circuits and systems with Universität zu Lübeck, Germany. From 2012 to 2014, he was the substitute of the Chair of Embedded Systems with Technische Universität Dresden, and the Chair of Computer Engineering with Technische Universität at Hamburg, Harburg, Germany. In 2015, he was appointed as a Professor with the Chair of Organic Computing with Universität zu Lübeck, with research focus on adaptive digital systems. Since 2016, he has been the Chair of Hardware-Oriented Technical Computer Science with Otto-von-Guericke Universität Magdeburg, Germany. His research interests include on-chip communication architectures, heterogeneous 3D SoC designs, methodologies for systematic design space exploration, and the runtime management of heterogeneous system architectures.