Communication Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 345–351
DOI: 10.15439/2023F6208
ISSN 2300-5963 ACSIS, Vol. 37

A modular and verifiable software architecture for interconnected medical systems in intensive care

Marc Wiartalla, Frederik Berg, Florian Ottersbach, Jan Kühn, Mateusz Buglowski,
Stefan Kowalewski, André Stollenwerk
Informatik 11 - Embedded Software, RWTH Aachen University
Ahornstraße 55, Aachen, Germany

***Abstract*—Medical device interoperability enables new therapy methods and the automation of existing ones. Due to different medical device manufacturers and protocols, we need auxiliary hardware and software for the interconnection. In this paper we propose a service-oriented software architecture built on a real-time operating system in order to create a modular medical cyber-physical system consisting of networked embedded nodes. In particular we highlight the need for the application of formal methods to ensure the functional safety of the system.**

I. Introduction

IN MEDICAL intensive care many different medical devices from various manufacturers are used for therapies. More and more future therapy methods are based on interconnected devices, called medical cyber-physical systems or cyber-medical systems [1], [2]. One particular case is the class of physiological closed loop control systems that aim to control one or several physiological parameter based on sensor measurements. These systems enable new therapies and the automation of existing ones and thus relieve the clinic personnel. In such systems flexibility and modularity is essential. Firstly, because clinics use different medical devices by manufacturers with different interfaces and protocols. Secondly, because it is always necessary to react to new insights or changes in the patient's condition, e.g. extend the therapy with more devices.

However due to the current legislation, a medical device which consists of an interconnection of different medical devices that are already authorized for the market, needs to pass the complete authorization process again. Hence, today many of these devices are interconnected manually by clinic personnel, e.g. reading measurements from one device and feeding these values into another device. Of course, for the future this legislation issue is supposed to be solved. Once this interconnection of authorized medical devices is integrated in the legislation, we need a software architecture to allow for the interconnection of these devices in a safe manner. As it is necessary to react to new insights in an agile way during therapy, the software architecture needs to be reliable and modular.

Without a doubt, medical systems are safety-critical as any fault can lead to a patient's harm or even death. In our opinion, it is therefore essential to apply existing state-of-the-art formal methods for the verification and validation of the medical software. Even though the use of formal methods is recommended by authorities, the application is still not enforced today [3]. Our thesis is that a future software architecture must support verification as best as possible.

In the use-case of physiological closed loop control systems the reaction time to external events like a change in physiological parameters is crucial. It might also be necessary to fulfill real-time constraints. This is why the vast majority of these algorithms are executed on embedded devices. Instead of using one centralized complex system, this allows us to distribute the system among multiple smaller nodes. Thus, resulting in significantly reduced software complexity on each node without reducing the overall processing capabilities. In this paper we present a modular extension of an existing real-time operating system in order to create a medical cyber-physical system consisting of networked embedded nodes.

A. Worked example

As a worked example for the next chapters we use the automation of extracorporeal membrane oxygenation (ECMO), where a patient with severe lung failure is supported by blood-based gas exchange outside the patient's body [4]. Figure 1 shows a sketch of the setup.

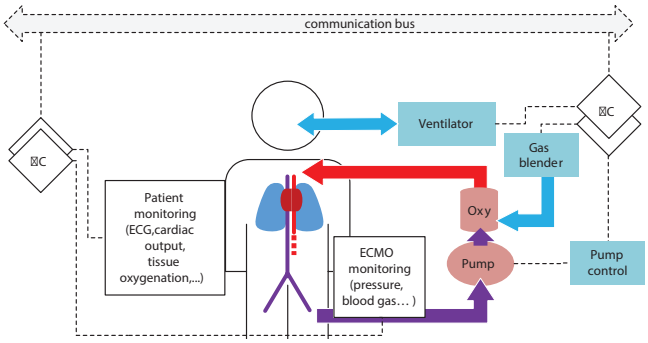

Fig. 1. Worked example: Automated ECMO therapy setup [5]

In the setup, a blood pump creates a blood flow through an oxygenator. Within the oxygenator, there is a blood and gas flow separated by a special membrane. This membrane allows for gas exchange, in particular the enrichment of blood with oxygen and the elimination of carbone-dioxide out of the blood. As a second actuator besides the blood pump, the gas-blender is in charge of controlling the gas composition and flow through the oxygenator. The sensors in the worked example are an online blood gas monitoring system as well

Thematic track: Software Engineering for Cyber-Physical Systems

as different pressure and flow sensors. In this setup we also implemented an interface for the patient's ventilator to efficiently cover the interaction between both therapy devices.

II. State of the Art

For the related work, we will analyze three aspects. First, we will present existing real-time operating systems (RTOS), on which our software architecture is built. Secondly, we will give an overview over the existing messaging protocols for the communication between nodes. Finally, we will present several projects working in the field of medical device interoperability. With these three aspects in mind we can highlight the key differences to our proposed architecture. In addition, we will present related work from non-medical domains.

The basis for our software architecture is the operating system underneath. For safety-critical embedded systems, the preferred option are real-time operating systems for microcontrollers due to the resulting determinism. The market-leading real-time operating system is FreeRTOS [6], which includes a kernel and Internet of Things (IoT) libraries. An alternative operating system is ChibiOS [7], which includes an operating system as well as a Hardware Abstraction Layer (HAL), peripheral drivers and a complete development environment. Furthermore, there exist several real-time variants of the Linux operating system, which are designed for the use in safety-critical software but are far more complex than the previously listed operating systems for microcontrollers [8]. These Linux operating systems might also contain pre-compiled libraries without source-code access.

Next, we will discuss which messaging protocols exist for the data exchange between nodes. Here we will focus on Ethernet for the communication between nodes. Many current research projects use the data distribution service (DDS) standard for interconnectivity, standardized by the object management group (OMG) [9]. DDS allows participants to communicate by publishing and subscribing to topics. The data can be shared with Quality of Service (QoS) specifications to ensure certain properties like reliability or periodicity. While DDS is an Application Programming Interface (API) specification, there exist several different DDS implementations, for example the C++ implementation eprosima FastDDS [10] and embeddedRTPS [11], a portable and open-source implementation of the Real-Time Publish-Subscribe Protocol for embedded systems. There also exist several alternative messaging transport protocols such as the MQTT protocol [12], a lightweight publish/subscribe messaging protocol which is often used in the IoT.

After discussing the state of the art of the underlying operating system and messaging protocol, we will present related work in medical device interoperability. Together with the Center for Integration of Medicine & Innovative Technology (CIMIT), the Medical Device 'Plug-and-Play' Interoperability Program (MD PnP) proposed the open standards for the Patient-Centric Integrated Clinical Environment (ICE) [13]. The standard defines the conceptual model, general requirements and different clinical use cases. Additional standards like the data logger are planned and in work. In the MD PnP program, an open source implementation of an interconnection environment called OpenICE was developed [14]. In an OpenICE system distributed network nodes are connected via Ethernet. OpenICE Device Adapters act as bridges to connect medical devices with the network. In addition, a central OpenICE supervisor unit runs clinical applications and logs data. All nodes in an OpenICE system have to be Java-capable devices like Linux computers. OpenICE uses DDS as the messaging protocol, in the current version the DDS implementation by Real-Time Innovations (RTI) [15]. For the OpenICE architecture several supervisor application examples were implemented, e.g. the synchronization between a ventilator and the shutter of an x-ray as a closed loop control use-case.

In the project OR.NET [16] concepts for open medical device interoperability in the operating room and clinic were developed. The concept of a service-oriented medical device architecture (SOMDA) including the technical specification was standardized in the IEEE 11073 SDC family [17], [18], [19]. In the OR.NET project, the service-oriented device architecture (SODA) was refined to the SOMDA paradigm. Besides standardized interface descriptions, a standardized way to describe provided and exchanged data was developed. As a base technology, the Devices Profile for Web Services (DPWS) is used for communication. Several open source frameworks implement the IEEE 11073 SDC standards in different programming languages, e.g. openSDC (Java) and SDCLib/C (C++).

Another research project called Smart Cyber Operating Theater (SCOT) works on an integrated operating room [20]. The SCOT project is based on the ORiN network interface for robot systems [21].

From the previously mentioned projects, the ICE project proposes concepts for the complete clinical IT infrastructure and follows a more centralized approach, with a central ICE Supervisor unit, that runs all applications. This differs from our approach, where the system is distributed among multiple smaller nodes. The OR.NET and SCOT projects mostly focus on an integrated operating room. The focus on the operating room results in specific requirements, e.g. a centralized console for visualization, and many scenarios have no real-time requirements [16]. In addition, a major part of the OR.NET project is the standardization of interfaces, to specify which data are exchanged between devices.

In contrast to the related work, our proposed software architecture is aimed at medical systems in the intensive care unit. In this paper we also exclude the direct communication with the clinical network. Notably, physiological closed loop control systems are a special use-case as these are highly automated. The main difference is that the patient in an operating room is under constant human monitoring. This is not the case for the intensive care unit, where patients are in critical state but not always under direct monitoring by the staff. In the future, it is even conceivable that such closed loop control systems are used outside of the hospital without

constant monitoring by medical professionals. In these use cases, the safety of these systems is essential, as medical professionals can not immediately react to faults in the system. Our architecture therefore includes a dedicated safety layer for various safety measures.

Additionally, we emphasize that it is necessary to use formal methods in the development process and increase safety through various design decisions. However, the presented OR.NET SDC and OpenICE projects are built on conventional Java-capable operating systems like Windows and Linux. Due to the high complexity, the formal verification and testing of the whole Linux operating system is still an open research topic. With Windows the verification and testing are even harder, as the operating system is not open-source. In addition, the usage of the Java runtime environment further increases the complexity and abstraction. There exists a real-time specification for Java to be used in real-time software development, which changes the semantics of the scheduling and memory management [22].

In contrast, our proposed software architecture builds on a real-time operating system for embedded systems with a small code scope and open-source libraries. This operating system can be verified through formal methods as it is far less extensive and complex. In addition, we aid the usage of formal methods through certain design decisions presented in the later chapters.

The previously mentioned interoperability projects all focus on the medical environment. However, the interconnection of nodes and data exchange is also relevant in different non-medical domains. The robot operating system 2 (ROS 2) [23] is a middleware for building robot applications including software libraries and tools. The second version improved many limitations of ROS 1. Similar to our approach a ROS system consists of a network of nodes which communicate over DDS. However, the provided tools are specialized for robot applications, while our proposed software is aimed at medical applications. With ROS-Health an extension of the Robot Operating System was developed for the use in neurorobotics [24].

III. Embedded Software Architecture

The software architecture proposed for interconnected intensive care applications is based on a real-time operating system (RTOS) for embedded systems. In our architecture, we use different features of the operating system like multitasking, timers and synchronization. In particular, the operating system should include a hardware abstraction layer. This abstraction allows us to freely change the used hardware platform. As there exist several operating systems with these features, the specific operating system can be abstracted. Figure 2 shows the resulting software architecture.

The following chapter describes the proposed software architecture from the top to the bottom. First, the application layer is presented with different development methods for data processing algorithms. Next, the data provisioning layer with the used communication protocol is introduced. Finally, we will discuss how our software architecture supports the application of formal methods to ensure safety.

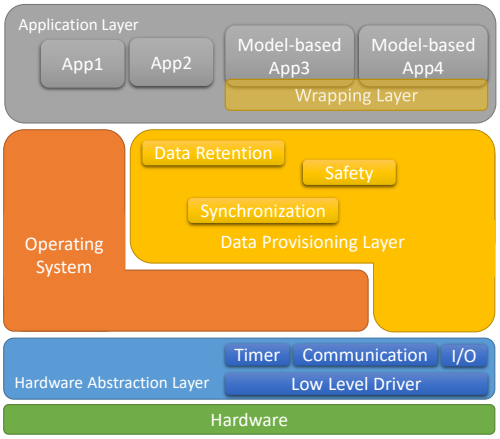

Fig. 2. Reference software architecture for medical applications in intensive care

A. Application Layer

Topmost in the architecture is the application layer. The applications are the actual data processing algorithms. In our architecture one node can run multiple applications depending on the requirements, e.g. the required computational power. Applications can also be moved between nodes without the need to adapt the application.

Applications can either be implemented directly or be generated from specific models. For model-based development we create models, e.g. diagram-based, using suitable software tools and expertise by medical personnel. From these models we can automatically generate code which is supported by several modern modeling tools. The generated code has to be connected with the data retention layer. Therefore we introduce a wrapping layer. If suitable physiological models of the patient or models of the physiological processes are available, we can also simulate the system and test our applications against these models.

As the software architecture is based on an embedded real-time operating system, each application has to be registered as a task by the developer with a declaration of the required stack size for this application. In addition, we do not use dynamic memory allocation or management offered by the operating system, in order to prevent memory allocation failures during runtime and aid the application of formal methods in such safety-critical systems.

B. Data Provisioning Layer

The main task of the data provisioning layer (DPL) is to store the data needed by the applications on the individual node. In addition, the data generated by this node needs to be communicated throughout the network. For this the concrete communication medium can be abstracted as it depends on the application requirements.

We implemented the data provisioning as a separate layer to enable modularity. This allows us free movement of applications between the different nodes in the system without the need to adapt the application. For this we create a global listing of all possible measurement and internally generated values in the system. This global listing is called communication matrix as we adopted the name from the Controller Area Network (CAN). The communication matrix contains unique identifiers for each message as well as additional information like label, description, scaling factor and unit. The wrapping layer and the DPL are automatically generated from this matrix, so the communication matrix will be referenced in the following architectural parts. In our architecture each application has to announce which data it uses, so that only the required data is stored on each node. This is later referenced as the selective part of the data provisioning layer.

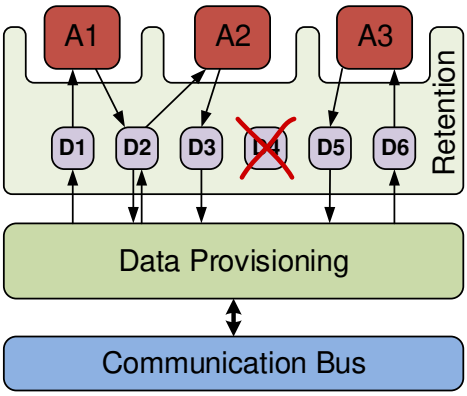

Fig. 3. Realization of the Data Retention [25]

Figure 3 shows the resulting data retention in our architecture. In this example the communication matrix consists of six different pieces of information (data) D1 to D6. Three different applications A1 to A3 run on the depicted node and read and write different data. As the data D4 is not used by any application it is locally eliminated. This methodology facilitates the movement of applications between nodes.

1) Time Synchronization: The simplest implementation for the data retention would be to just store the last broadcasted measurement value. However, in addition it is also necessary to store the timestamps of measurements and generated values. As an example, we describe two scenarios from the ECMO. First, we might want to keep track of a patient's body temperature during the treatment. To put the temperature measurements in a chronological order, we need timestamps for each measurement. However, in this scenario an accuracy of one second is sufficient. Second, it might be necessary to compute the patient's dynamic lung compliance during mechanical ventilation, which can be calculated from the tidal volume and pressure. For the calculation of the compliance we therefore need to correlate pressure measurements and flow measurements. Thus, we need precise measurement timestamps with an accuracy in the range of few milliseconds.

To support such scenarios, we need to keep track of time locally on each specific node, but also globally for the whole network of nodes. This global time allows us to check if a value is outdated but also correlate values from different nodes. The required accuracy of the time synchronization depends on the specific application requirements and used synchronization protocol.

For this we define a master-node and synchronize all other nodes to this node. There are several existing network synchronization protocols depending on the choice of the communication medium, for example OCS-CAN [26] for CAN and the Precision Time Protocol (PTP) [27] for Ethernet communication.

Overall the synchronization has to be defined in a way, such that the local clock-error on a specific node does not exceed an upper bound. In the example of PTP the accuracy depends on whether we use a hardware or software implementation. In our worked example we use a software implementation and can achieve an accuracy of 1 millisecond. Next, we present the data retention layer.

2) Data Retention: In medical applications it is not only necessary to work with single measurement values but also time series. In the calculation of the lung compliance during mechanical ventilation, we need to store series of flow and pressure measurements to compute the change in volume and pressure.

In order to support such medical scenarios, the data retention is able to store a time series of data. We can configure the length of the series as well as the sample rate. This can be configured for each individual app with reference to the used measurements. Since it is not always necessary to work with the complete series of data, we realized general and specific operations on these time series. These resulting values are constantly calculated in the background as new generated data is received. As general operators we implemented the minimum, maximum, median and mean operations. These should already address many needs during the processing of medical data. In addition, it is also possible to define specialized data operations tailored for specific applications. This is realized by allowing the user to register a callback function, which is executed on the time series.

3) Safety: As mentioned before, medical applications in particular are safety critical. Therefore, we include an additional safety layer in the software architecture [28]. In this layer, the control values for actuators can be safeguarded but also measurement values of sensors can be annotated with a metric regarding aspects like plausibility, data quality or age.

The basic approach to safeguard transmitted values are general boundaries defined in the communication matrix. Medical devices often have maximum ratings for their operation. In our worked example, the used oxygenator has a maximum rating for the set-values of gas and blood flow. Moreover, to maintain a basic patient support in the context of the treatment we might want to define a minimum gas and blood flow depending on the patient's parameters. In addition to this, even more intelligent safeguarding is possible. We can

derive mathematical equations from physical or physiological models and integrate them in the safety layer. Also the usage of artificial intelligence is possible. In a third step, it is also important to consider the age of a measurement value. If a measurement in our system is several minutes old, we have to react accordingly and for example, use a fallback value. Furthermore, we might need information about the quality of measured data, which can be transmitted over the network with additional messages if this information is available.

C. Formal Methods

Since software errors can lead to harm of the patient and even death, medical software is highly safety-critical. We therefore emphasize that the application of formal methods in the software development process is essential.

In our embedded software architecture, we made several design decisions, which aid the application of formal method. These are mainly the fully static architecture and knowledge of the underlying operating system. In the application layer we have to declare the stack size for each registered task. With this information we can then perform a stack size analysis to prevent stack overflows. Many compilers already offer stack size analysis capabilities [29] and from our point of view, this is one of the most important measures to safeguard a medical cyber-physical system.

Furthermore, static analysis enables us to prove the absence of critical run-time errors without having to execute the code. Common errors that can be found with static analysis techniques are dead or unreachable code, uninitialized variables, null pointer dereference and invalid arithmetic operations. It is advisable to use static analysis early in the development process as bugs are cheaper to fix compared to later development stages. [30]

The adherence and documentation of compliance with a suitable coding guideline, like MISRA C [31], MITRE Common Weakness Enumeration (CWE) [32] or the SEI CERT C coding standard [33] is highly recommended when dealing with safety-critical systems. The MISRA C coding guidelines define a subset of the C language without parts with e.g. undefined behavior. This is aimed at error prevention and is supposed to increase code readability and explainability [34]. The MISRA C rules can be checked by algorithms, which will be used in the worked example implementation.

IV. Worked Example Implementation

In our worked example, we implemented the automation of an extracorporeal membrane oxygenation therapy for patients who suffer from Acute Respiratory Distress Syndrome (ARDS) [35], [36]. In the setup, microcontroller nodes were connected to medical devices and communicate over a network or communication bus. Therefore, we implemented the software architecture as presented in the previous chapter. The implemented architecture with the used RTOS and hardware is shown in figure 4.

The implementation of the software architecture with an example project for an OxiMax N-560 pulse oximeter is

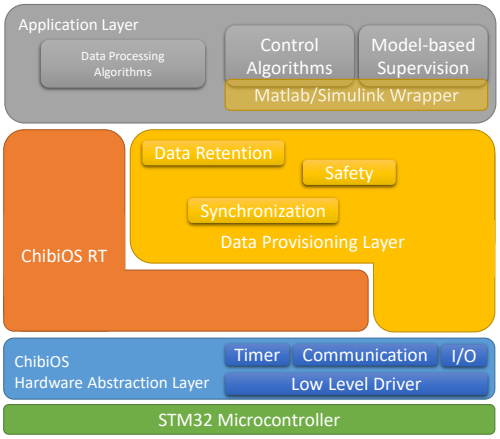

Fig. 4. Implementation of software architecture

available in [37]. As the real-time operating system we use ChibiOS as it offers a wide range of features including a fully static architecture and hardware support for the used STM32 microcontroller. Because of the existing hardware abstraction layer within ChibiOS, we were able to switch from our previous hardware platform with an Atmel AT91 microcontroller to a STM32 microcontroller with minimal changes in the software. The communication matrix is defined in multiple TOML (Tom's Obvious Minimal Language) files. Based on these TOML files we generate parts of the data provisioning and safety layer. The selective part of the data provisioning layer is based on the C compiler's pre-processor. Thus, we still only use static memory allocation.

In the first version of the architecture, the nodes were connected via CAN. In the latest version, we added Ethernet for the communication in order to allow better integration of mobile devices for monitoring.

We use an existing end-to-end middleware, because we have less code to maintain as the specification and different implementations already exist. We decided to use DDS, as it offers us great flexibility to control the communication behavior through the quality of service policies and has an integrated discovery system. With these QoS policies we can enforce additional safety requirements like the periodicity of measurements or the usage of redundant sensors. If the periodicity of data is known, we can notify the user about missing measurements. In addition, DDS is already established in several safety-critical scenarios, like aviation or military applications [38]. Another advantage is that DDS is suited for the use in embedded systems and offers a wide language and platform support, which eases the integration of other devices. For the specific DDS implementation, we use embeddedRTPS [11] as it is suited for embedded systems.

In our worked example implementation, it is possible to prioritize certain messages like alarms, as these should be treated with higher priorities. These alarm message have to be prioritized in the software as well as in the network. Using Ethernet, these messages are prioritized with the differenti-

ated services code point (DSCP) in the IP header. However, we also need infrastructure, like switches, that support this prioritization. For CAN communication the CAN message identifier determines the priority with a low message identifier representing a high priority.

The actual applications, e.g. control algorithms, are either implemented directly in the language C or generated C-Code from Matlab Simulink (The Mathworks, Natick, MA) [39] models is used. We automatically generate Simulink blocks from the TOML files to allow the developer to use any data out of the communication matrix without the need to take care of its retention.

In the described implementation, we solely use static memory structures. Additionally, we apply different formal methods. First of all, we carry out a stack size analysis and compare the results with the declared stack size registered to the operating system. In addition, we carried out static analysis using Polyspace (The Mathworks, Natick, MA) [40] to prove the absence of critical run-time errors. Furthermore, we used Polyspace to check the compliance of our software with the MISRA C rules.

Finally, we also carried out a worst-case execution time analysis of the used algorithms to get an over-approximation of the total CPU consumption on a specific node in the medical cyber-physical system. The analysis was conducted using aiT from AbsInt [41]. One analyzed algorithm is the model-based blood pump supervision, which can detect and predict events like the suction of the withdrawing cannula to the wall of the surrounding vessel or the presence of gas bubbles in the blood tubing. This analysis leads to a worst-case CPU time of 0.138 milliseconds on the Atmel AT91SAM7 hardware, which makes the algorithm suited for the use in an embedded environment. [36]

V. Results

In section III, we presented a software architecture for interconnected medical systems. The architecture enables modularity, as we can move applications between nodes depending on the required memory and CPU time. However, the network delay has to be considered. We might prefer to process sensor data on the same node as the control algorithm, which uses this data as it's input. Through a loopback mechanism this data does not need to be sent over the network, which decreases the delay.

Additionally, the data provisioning layer provides basic safety mechanism which can be easily extended with more complex methods like physiological models. This goes hand in hand with the need for precise global timestamps of measurements or information on the time passed between measurements. This enrichment of the bare data with additional information also helps with the safeguarding of measurements. The safety layer can be extended depending on the requirements, however, often the difficulty is to find physiological models in the appropriate abstraction to derive mathematical formulas. In addition, we highlighted the need for formal methods in the development process and aided the application by various design decisions.

One major advantage of the presented software architecture is the free choice for the used hardware and operating system. Because of the abstraction layer, we do not need to modify our existing software when enhancing an already existing setup.

VI. Conclusion

In this paper, we presented a modular and verifiable software architecture based on a real-time operating system suited for interconnected medical systems in intensive care environments. We presented the general concept of our proposed entities, of which such a medical cyber-physical system can consist of, and presented a worked example implementation for the automation of an extracorporeal membrane oxygenation therapy. The main component we introduced was the generated data provisioning layer and its interactions to the other parts of the software architecture. The DPL takes care of the data storage, needed by the different applications and algorithms. Additionally, the DPL offers time synchronization as well as communicating the data between nodes. The included safety layer allows us to safeguard transmitted values and measurements. Finally, we enable the integration of model-based generated code into this software architecture by a wrapping layer.

We presented a strictly static software architecture, which allows for the efficient use of formal methods. This allows us to prove the absence of possibly safety related issues like the violation of real-time constraints or memory/stack overflows, which need to be avoided in medical systems. In conclusion, the combination of these measures allows for the safe operation of medical cyber-physical systems.

However, the applied formal methods presented in this work are just the standard methods and the verification of more properties is possible. A next step would be the validation and verification of the control algorithms to ensure the patient's safety. Though, the verification in a physiological closed-loop system requires an accurate physiological model of the system, which is challenging due to the complexity and variability of the human physiology.

As an further outlook, we plan to improve the safety layer by deriving safeguarding algorithms out of publicly available physiological models. Through the use of publicly available databases, there is also a high potential to automatically learn the needed relations between different physiological values by means of artificial intelligence. In addition, we plan to add different methods for the detection and diagnosis of hardware faults and medical complications in the system.

VII. Acknowledgments

The authors gratefully acknowledge the funding of the German Research Foundation (PAK 138/1 and 2) and the Federal Ministry of Education and Research (031L0253B), which allowed for these results.

REFERENCES

[1] I. Lee and O. Sokolsky, "Medical cyber physical systems," in *Design Automation Conference*, pp. 743–748, ACM, 2010.

[2] G. De Micheli, "Cyber-medical systems: Requirements, components and design examples," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2226–2236, 2017.

[3] S. Bonfanti, A. Gargantini, and A. Mashkoor, "A systematic literature review of the use of formal methods in medical software systems," *Journal of Software: Evolution and Process*, vol. 30, no. 5, p. e1943, 2018.

[4] Rüdger Kopp, Ralf Bensberg, Marian Walter, Jutta Arens, Rolf Rossaint, and André Stollenwerk, "Automation of extracorporeal membrane oxygenation using a combined safety and control concept.," *Intensive Care Medicine*, vol. 37, no. S1, 2011.

[5] J. Kühn, C. Brendle, A. Stollenwerk, M. Schweigler, S. Kowalewski, T. Janisch, R. Rossaint, S. Leonhardt, M. Walter, and R. Kopp, "Decentralized safety concept for closed-loop controlled intensive care," *Biomedical Engineering/Biomedizinische Technik*, vol. 62, no. 2, pp. 213–223, 2017.

[6] Richard Barry, "FreeRTOS," 2023. https://www.freertos.org/.

[7] Giovanni Di Sirio., "ChibiOS," 2023. https://www.chibios.org.

[8] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–36, 2020.

[9] Object Management Group, "Data distribution service specification, version 1.4," 10.04.2015.

[10] eProsima, "Fast DDS," 2023. https://www.eprosima.com/index.php/products-all/eprosima-fast-dds.

[11] A. Kampmann, A. Wustenberg, B. Alrifaee, and S. Kowalewski, "A portable implementation of the real-time publish-subscribe protocol for microcontrollers in distributed robotic applications," in *The 2019 IEEE Intelligent Transportation Systems Conference - ITSC*, (Piscataway, NJ), pp. 443–448, IEEE, 2019.

[12] OASIS MQTT Technical Committee, "MQTT, Version 5.0," 07.03.2019.

[13] ASTM, "Medical devices and medical systems - essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ice) - part 1: General requirements and conceptual model," 2013.

[14] J. Plourde, D. Arney, and J. M. Goldman, "OpenICE: An open, interoperable platform for medical cyber-physical systems," in *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2014)*, (Piscataway, NJ), p. 221, IEEE, 2014.

[15] Real-Time Innovations, "Connext DDS," 2023. https://www.rti.com/.

[16] M. Kasparick, M. Schmitz, B. Andersen, M. Rockstroh, S. Franke, S. Schlichting, F. Golatowski, and D. Timmermann, "OR.NET: a service-oriented architecture for safe and dynamic medical device interoperability," *Biomedizinische Technik. Biomedical engineering*, vol. 63, no. 1, pp. 11–30, 2018.

[17] IEEE Engineering in Medicine and Biology Society, "IEEE Standard - Health informatics – Point-of-care medical device communication - Part 10207: Domain Information and Service Model for Service-Oriented Point-of-Care Medical Device Communication," 2017.

[18] IEEE Engineering in Medicine and Biology Society, "IEEE Standard - Health informatics – Point-of-care medical device communication - Part 20702: Medical devices communication profile for web services," 2016.

[19] IEEE Engineering in Medicine and Biology Society, "IEEE Standard - Health informatics – Point-of-care medical device communication - Part 20701: Service-Oriented Medical Device Exchange Architecture and Protocol Binding," 2019.

[20] J. Okamoto, K. Masamune, H. Iseki, and Y. Muragaki, "Development concepts of a smart cyber operating theater (scot) using orin technology," *Biomedizinische Technik. Biomedical engineering*, vol. 63, no. 1, pp. 31–37, 2018.

[21] M. Mizukawa, H. Matsuka, T. Koyama, T. Inukai, A. Noda, H. Tezuka, Y. Noguchi, and N. Otera, "Orin: open robot interface for the network - the standard and unified network interface for industrial robot applications," in *SICE 2002*, (Tōkyō), pp. 925–928, SICE, 2002.

[22] G. Bollella and J. Gosling, "The real-time specification for java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

[23] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[24] G. Beraldo, N. Castaman, R. Bortoletto, E. Pagello, J. del R. Millan, L. Tonin, and E. Menegatti, "Ros-health: An open-source framework for neurorobotics," in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)* (H. Kurniawati, ed.), (Piscataway, NJ), pp. 174–179, IEEE, 2018.

[25] A. Stollenwerk, F. Göbe, M. Walter, J. Arens, R. Kopp, and S. Kowalewski, "Smart Data Provisioning for Model-Based Generated Code in an Intensive Care Application," in *3rd Joint Workshop On High Confidence Medical Devices, Software, and Systems & Medical Device Plug-and-Play Interoperability : HCMDSS/MDPnP 2011 ; in conjunction with CPSweek 2011*, (Chicago), HCMDSS/MDPnP 2011, Apr 2011.

[26] G. Rodriguez-Navas, S. Roca, and J. Proenza, "Orthogonal, fault-tolerant, and high-precision clock synchronization for the controller area network," *IEEE transactions on industrial informatics*, vol. 4, no. 2, pp. 92–101, 2008.

[27] IEEE Instrumentation and Measurement Society, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," 2019.

[28] J. Kühn, A. Stollenwerk, C. Brendle, T. Janisch, M. Walter, R. Rossaint, S. Leonhardt, S. Kowalewski, and R. Kopp, "Sensor supervision and control value limitations in networked intensive care," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016* (W. Zimmermann, L. Alperowitz, B. Brügge, J. Fahsel, A. Herrmann, A. Hoffmann, A. Krall, D. Landes, H. Lichter, D. Riehle, I. Schaefer, C. Scheuermann, A. Schlaefer, S. Schupp, A. Seitz, A. Steffens, A. Stollenwerk, and R. Weißbach, eds.), vol. 1559 of *CEUR Workshop Proceedings*, pp. 187–194, CEUR-WS.org, 2016.

[29] E. Botcazou, C. Comar, and O. Hainque, "Compile-time stack requirements analysis with gcc: Motivation, development, and experiments results," in *Proc. GCC Developers Summit*, pp. 93–105, 2005.

[30] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*, pp. 581–591, Springer, 2015.

[31] A. Burnard, P. Burden, L. Whiting, C. Tapp, G. McCall, M. Hennell, C. Hills, and S. Montgomery, "MISRA C:2012," 2013.

[32] P. Anderson, B. Curtis, P. Braione, A. Summers, C. Eng, J. Fung, J. Gazlay, A. Hoole, J. Jarzombek, J. Lam, C. Levendis, J. Oberg, K. Seifried, C. Turner, and A. van der Stock, "Common weakness enumeration," *Mitre Corporation*, 2007.

[33] Software Engineering Insitute CERT, "C coding standard: Rules for developing safe, reliable, and secure systems," *Reliable, and Secure Systems*, 2016.

[34] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software," in *Static Analysis* (A. Podelski, ed.), vol. 11002 of *Lecture Notes in Computer Science*, pp. 5–23, Cham: Springer International Publishing, 2018.

[35] R. Kopp, R. Bensberg, A. Stollenwerk, J. Arens, O. Grottke, M. Walter, and R. Rossaint, "Automatic control of veno-venous extracorporeal lung assist," *Artificial organs*, vol. 40, no. 10, pp. 992–998, 2016.

[36] A. Stollenwerk, J. Kühn, C. Brendle, M. Walter, J. Arens, M. N. Wardeh, S. Kowalewski, and R. Kopp, "Model-based supervision of a blood pump," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 6593–6598, 2014.

[37] M. Wiartalla, F. Berg, F. Ottersbach, J. Kühn, M. Buglowski, S. Kowalewski, and A. Stollenwerk, "A modular and verifiable software architecture for interconnected medical systems in intensive care," 2023. https://doi.org/10.18154/RWTH-2023-07342.

[38] Object Management Group, "Who's Using DDS?," accessed 21.12.2022. https://www.dds-foundation.org/who-is-using-dds-2/.

[39] The Mathworks, Inc., "MATLAB Simulink (R2022b)," 2022.

[40] The Mathworks, Inc., "Polyspace (R2022b)," 2022.

[41] C. Ferdinand, "Worst case execution time prediction by static program analysis," in *Proceedings / 18th International Parallel and Distributed Processing Symposium*, (Los Alamitos, Calif.), pp. 125–127, IEEE Computer Society, 2004.