On the Model Continuity in Control Systems Design Using DEVS, UML, and IEC 61499

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology,
Bozetechova 2, 612 66 Brno, Czech Republic
email: {koci,janousek}@fit.vut.cz

***Abstract*—This paper deals with various ways of design and implementing Distributed Control Systems (DCS). The authors are familiar with the Unified Modeling Language (UML), the Discrete Event System Specification (DEVS) formalism, the IEC 61499 standard, and different tools, such as 4diac, Node-RED, or PowerDEVS. The paper presents the basics of methodology for the design and implementation of control software, which will allow the use of any of these approaches following a conceptual model based on UML. The main reason is to describe the whole development process, from the standard UML models to their implementation, and enable the developers to use modeling or design techniques they are familiar with. The promising way is to apply the model-continuity principle in conjunction with the DEVS formalism, which can be used as a unifying platform for design, and in some cases, as the most appropriate path to implementation. Using DEVS, we get the possibility of directly applying the simulation during the design, thus a more straightforward validation of the proposed system. The DEVS model can also be transformed into alternative implementation models. The considered principles is demonstrated on a case study of a simple system, Central heating with zone valves.**

***Keywords*—Control systems; IoT; UML; DEVS; IEC 61499; simulation; model continuity.**

I. Introduction

The paper presents the basics of methodology for the design and implementation of control systems. We consider the tree-level structure of such a system: sensors/actuators, distributed controllers, and a Supervisory Control And Data Acquisition (SCADA) system. The standard IEC 61499 addressing function blocks for industrial processes and control systems was established in 2005. It defines a generic model for distributed control systems. Various environments and tools follow IEC 61499 principles, e.g., 4diac with the corresponding runtime environment FORTE [1]. Different approach to the implementation of control systems or their parts is represented by, e.g., Node-RED [2]. Nevertheless, a uniform procedure including a problem analysis or requirements specification is not defined. The motivation for this work is to describe the whole development process, from the conceptual models to their implementation following the Model-Driven Development (MDD) and model-continuity principles, and allow any of the commonly available approaches. Our goal is to define and use a unified approach independently of the target implementation environment.

MDD is a development methodology that supports the use of models as significant artifacts [3]. The development process is a series of constant refinement and transformation of models. Ideally, more specific models are generated and, in the last step, the code for a particular platform. Unified Modeling Language (UML) is a standard language for modeling various aspects of software systems, both in academia and in industrial development, thanks to a sophisticated graphical representation. However, the ability to simulate and investigate models in real conditions limits the use of UML [4]. It is, therefore, appropriate to look for proper formalisms or approaches that can build on UML models and simplify the simulation and transition from models to implementation.

A typical example of a combination of formal modeling and simulation is the Discrete Event System Specification (DEVS) [5]. DEVS is a modular and hierarchical formalism for modeling and simulation of discrete event systems, systems of differential equations (continuous systems), and hybrid systems. DEVS models can be interconnected through input/output ports to create modular and hierarchical topologies of blocks. In the context of modeling and development of control systems, the Model Continuity for DEVS has been introduced [6][7]. The main idea of model continuity is that a DEVS simulation model for a controller can evolve during the development process from a pure simulation (in a simulated environment) until its final deployment in the target environment without re-implementation. The DEVS simulation engine becomes a run-time execution environment for the target system. This approach leads to the fact that no errors are introduced into the target implementation during the development.

For our work, we chose DEVS because it is well-defined, intuitive, understandable, and universal. DEVS-based real-time simulation engine (in our case PowerDEVS [8]) can be used in the role of runtime execution environment. In addition, we also consider the possibility of using other environments for implementation. These environments interpret DEVS similar models such as Node-RED flows or IEC 61499 applications (in our case, we use the open-source development environment 4diac with the runtime environment FORTE).

The paper is structured as follows. We discuss related work in Section II. Section III addresses the conceptual modeling of control systems, introducing a case study. In Section IV, we describe the possibilities of creating the Platform Independent Model (PIM) using UML and DEVS. Subsequently, in Section V, we will show the ways of creating the Platform Specific

 ISBN: 978-1-61208-894-5

Model (PSM) from DEVS PIM. Finally, in Section VI, we will show the possibility of extending to a distributed environment.

II. Related Work

Similar works are dealing with control system design using UML and the IEC 61499 standard. Many of them, e.g., [9]–[11], propose generating IEC 61499 from UML, or System Modeling Language (SysML), typically from a class diagram. Other works, such as [12], deal with the behavior of atomic components and propose a transformation of activity diagrams to the IEC 61499 execution control charts (ECC). Unlike these works, which interconnect UML and IEC 61499, we propose a step from UML to DEVS during the development process.

Other approaches, e.g. [13], [14] deal with the behavioral model of atomic components. In contrast, we focus only on the structure of components and sub-components in this paper. We assume the availability of a library of well-defined atomic components in the target environment.

There are also approaches that attempt to transform conceptual models, such as those described by SysML, into simulation models [15]. In contrast, we do not consider the simulation model a goal but a potential option in the development. Our main goal is a unified methodology leading to the implementation of a control system in different environments. The novelty of our approach is that we use DEVS as a basis for various ways of implementation. DEVS, IEC 61499, and Node-RED execution environments are presented as examples.

III. Conceptual model of control system

In this part, we will present the basic approach to creating conceptual models using the UML language and then a simple example (case study), on which we will demonstrate our approach to the design of control systems.

A. UML in Control Systems Design

UML is an acknowledged and used language for conceptual design of software systems, including control systems. Their advantage is that they can offer different views of the same system to get a complete overview of the complex system. Now, we briefly recall the basic way of designing software systems using the UML language. Although UML itself does not define the methodology and assumes that different approaches will use it, it is possible to identify basic processes, activities, and models applied to most design methodologies.

- *Requirements specification.* In the context of software design, we could also use the term system usage model, as a use case diagram is often used.
- *Structure specification.* To capture the structural elements of the system, which then contribute to the solution of individual use cases. Class, component and package diagrams are commonly used.
- *Behavior specification.* Diagrams capturing the behavior of classes (objects), use cases, or components. The most commonly used ones are activity diagrams (for use cases), state-charts, and interaction diagrams.

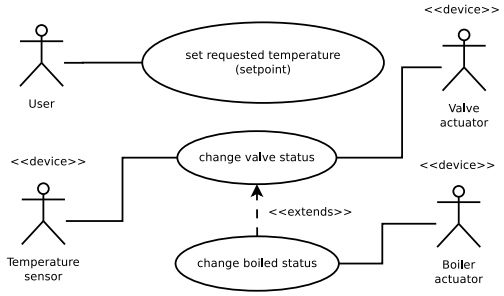

Figure 1. Use case model of the Case study.

The models designed in this way can be subjected to a more thorough analysis. Nevertheless, UML, including the object-oriented approach, retains features that are advantageous in the design of control systems — complexity, the ability to capture and formalize control system requirements, or the availability of tools. So, the full use of the object-oriented approach in the practical design and implementation of control systems was too demanding and expensive. From these reasons, standards for implementing control systems in the automation domain, e.g., IEC 61499, have gradually emerged. However, even in these approaches, UML models can be used to specify requirements and domain concepts (*conceptual modeling*), which can either be further developed by other paradigms, used as part of a simulation, or transformed into a language or environment that better suits the needs of the control system [16].

B. Case Study

We will use a simple example (case study) of Central heating with zone valves to demonstrate our approach. Each zone contains a temperature sensor, valve actuator allowing On/Off control, and Human-Machine Interface (HMI) capable of displaying all relevant data points and setpoint adjusting. The zone controller compares sensor data and setpoint and decides whether to open or close the radiator valve. In our example, we consider two zones. Zone controllers are connected to the central controller. The central controller sets the boiler On/Off according to the state of zone valves; if and only if at least one of the valves is open, the boiler is instructed to heat. It also contains interface to central HMI/SCADA.

C. UML Conceptual Model of the Case Study

The first step in creating conceptual models is defining the system's requirements. As already mentioned, a use case diagram from UML is usually used. The initial overview is in Figure 1. We have identified four actors who interact with the system and are always involved in specific use cases. The actor can be a person (*User*), but also other systems or hardware components. We have identified three types of these components in our example—*temperature sensor*, *valve actuator*, and *boiler actuator*. The primary use cases are then the temperature setting by the user and the change of the valve setting (open/closed) based on the change of the

 ISBN: 978-1-61208-894-5

sensor temperature. This case can cause a change in the boiler settings, which is modeled by the extension case.

Another important model is the domain model. It captures the system's conceptual elements (classes) that are needed to solve individual use cases. The primary domain model is shown in Figure 2. The model is divided into two parts. The first part models the zone (*ZoneController*), and the second is the central unit (*CentralController*). In each part, we have identified the basic concepts of the system – sensors, actuators, the user interface (HMI) representation, and the basic controller. These parts communicate with each other, as indicated by the association between the *CentralCore* and *ZoneCore* classes.

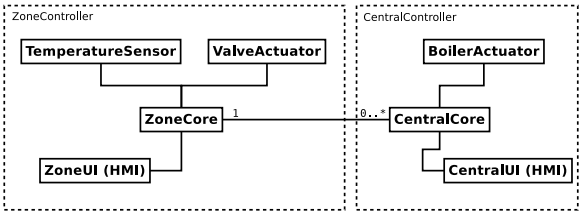

Figure 2. Basic domain model of the Case study.

Of course, conceptual classes are not sufficient for a more detailed description of the structure and behavior of the system; they serve primarily for the initial identification of concepts in the system design. In a further development, more detailed design models are created based on these concepts and use cases. It will be taken into account in Section IV.

IV. Platform Independent Model

The Platform Independent Model (PIM) is used to design a detailed system structure and behavior regardless of the specific platform on which the system will run. For the creation of PIM, modelling techniques should be used that allow simple automated transformation into PSM. We will continue with UML models. To use them as a starting point for further generation, we will mainly use the component diagram, because components are close to the concept of control software design. We will show that a similar effect can be achieved with the DEVS formalism, which can be directly simulated with a suitable tool and considered an actual control system model and its implementation.

A. UML Component Model of the Case Study

The UML component is a specialization of a structured class and as such forms a hierarchical model. It can be understood as an entity encapsulating a more complex structure of classes (and thus other components) and interchangeable with another component that meets the required interface.

Figure 3 shows the component corresponding to the *ZoneController* sets of conceptual classes. The component works with representations (classes or interfaces) for the temperature sensor, valve actuator and user interface (HMI), and contains the *ZoneCore* class implementing the decision algorithm. The component has one input and one output

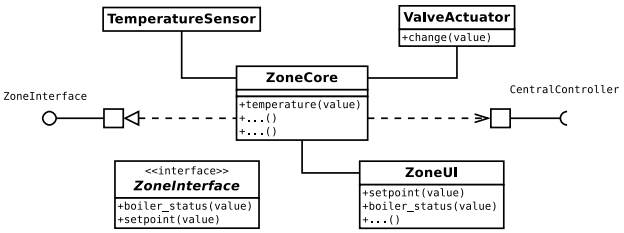

Figure 3. Component model of the ZoneController conceptual class.

port, each associated with an interface. The component offers the interface (provided interface, in the UML terminology) *ZoneInterface*, through which it receives external events, and requests the interface (required interface, in the UML terminology) *CentralInterface* from the connected components.

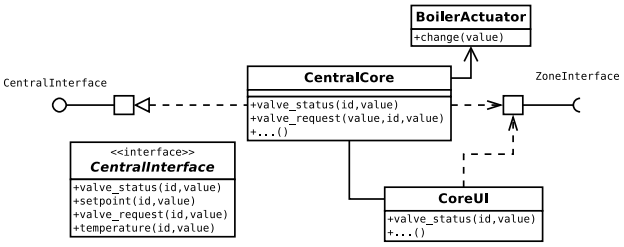

Figure 4. Component model of the CentralController conceptual class.

Figure 4 shows the component corresponding to the *CentralController* sets of conceptual classes. The component is structured similarly to the *ZoneController*. The component has the provided interface *CentralInterface* and the required interface *ZoneInterface*. Each event corresponds to the method of the respective interface. We assume that only simple data (temperature, on/off, etc.) is passed between the system elements, which can be annotated—the method attributes model annotations. In the example, only the identification of individual zones is used.

B. From UML Component Model to the DEVS

From the control systems design point of view, it is better, especially for clarity, that each event is associated with an individual port of the component. This is achieved by deriving specific interfaces from the original interface in the component diagram, following the principle of interface separation, where each such interface contains only one method corresponding to the event. The modified component model is shown in Figure 5.

It is possible to create a composite component that consists of other interconnected components. Figure 6 shows such the composite component for a simple system consisting of one zone controller. Simple, so-called atomic components, which are no longer further divided into internal parts, can be described by some of the behavior models, or another suitable formalism can be attached to them.

Finally, we can create the same component diagram using the DEVS formalism. Compared to the UML component

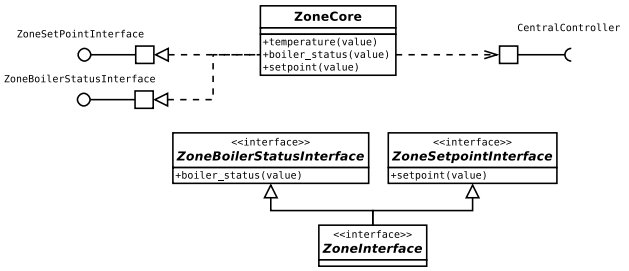

Figure 5. Part of the ZoneController component model having individual ports for each event.

model, which captures the system's static structure, the DEVS (component) model represents specific elements (objects) of the system. Thus, it represents both the model and the implementation of a specific system. In Figure 7, we see a DEVS model for a system with one central control and two zones. These components can be imagined as instances of the corresponding components in Figure 6, but they also define the structure.

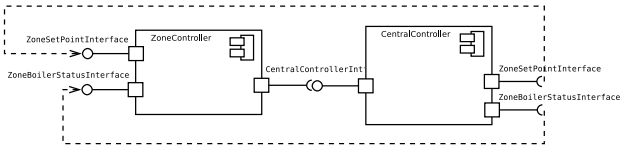

Figure 6. System component model.

It is then possible to derive systematically implementation for a specific platform, like PowerDEVS Node-RED, and 4diac/FORTE. In our approach, we understand this as a straightforward implementation of the Model-continuity principle [6]. It follows that in the case of DEVS, the PIM and PSM models merge–they differ mainly in the implementation of atomic components. Therefore, the DEVS models of the case study will be further discussed in Section V.

V. Platform Specific Model

In this section, we will present various ways of control system implementation, which are based on DEVS PIM models.

A. PowerDEVS implementation

The transformation of the DEVS model for a specific version of the DEVS platform is straightforward (various implementations differ only in detail). We use PowerDEVS [8] because it enables real-time simulation and can serve as runtime environment for DEVS models deployment.

The control model is in Figure 7. It contains two components, modeling zone controllers and one component, modeling the central controller. The interconnected component models the transfer of data between components (in DEVS terminology, these are events).

The model of the central controller is in Figure 8. The *or* component is responsible of deciding to keep the boiler on if and only if at least one of the zone controllers requires to

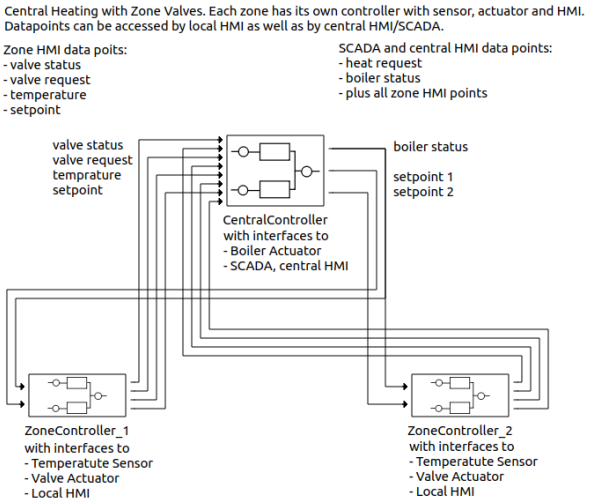

Figure 7. PowerDEVS model of the system.

heat. In addition to the basic functionality it also includes an interface on central HMI/SCADA.

By comparing with the domain model (see Figure 2) and the component model (see Figure 4), we find that the *Boiler_Actuator* component models the *BoilerActuator* concept, the *HDMI_SCADA_Pub_Sub* component represents the *CentralUI* user interface, and other components and their interconnections model the decision-making algorithm (*CentralCore*). The interface is then made up of ports and data transfer instead of interfaces and method calls.

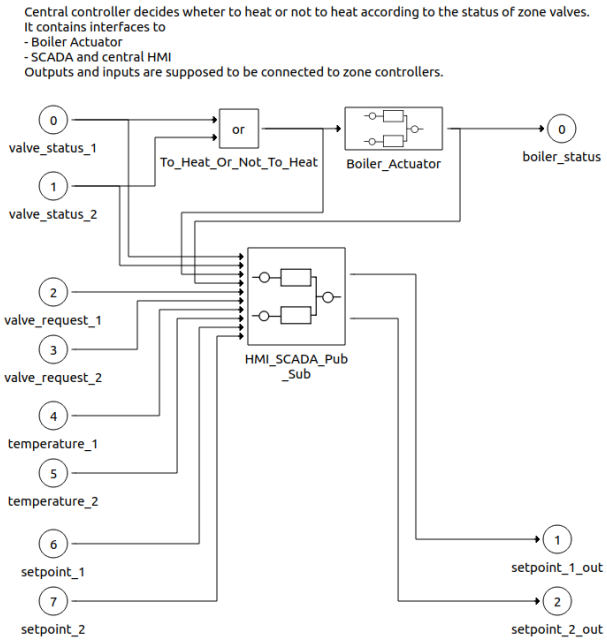

Figure 8. PowerDEVS model of central controller.

The zone controller model is shown in Figure 9. It contains an interface to a temperature sensor (in our case, it is a client of MQTT broker, which subscribed messages from the

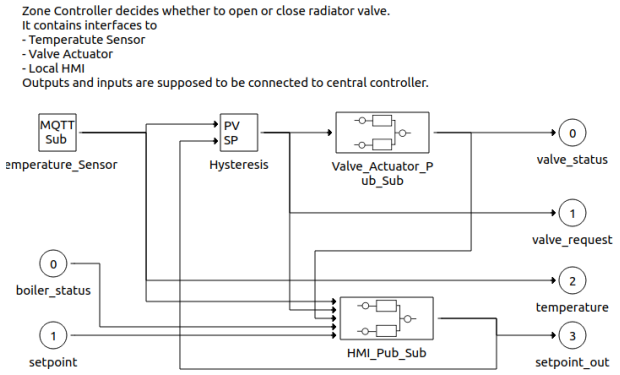

Figure 9. PowerDEVS model of zone controller.

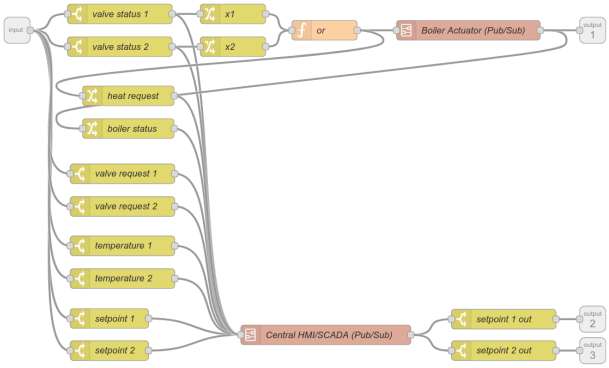

Figure 11. Node-RED model of central controller.

required temperature sensor), comparison with the setpoint available from the HMI (connected via MQTT pub-sub client too) is performed by the hysteresis component. The output is connected to the actuator interface (again via MQTT pub-sub component). The inputs and outputs of the zone controller enable the connection and transfer of data from/to the central controller. These data are used to operate the potentially distributed control and for presentation to the user via the HMI. Setpoints, i.e., required temperature values in zones, can also be set via local HMIs or the central HMI.

B. Node-RED implementation

Node-RED [2] is a popular tool for coordinating and automating the IoT systems. It is a platform for modeling and interpreting so-called flows, which is a concept similar to DEVS. DEVS is thus easily transformable into Node-RED.

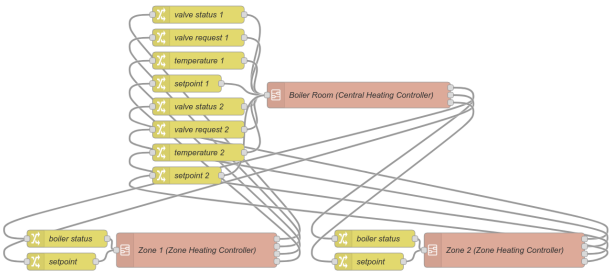

Figure 10. Node-RED model of the system.

The difference between Node-RED and DEVS is that it is not intended for simulations nor hard real-time applications. However, for implementation of soft real-time applications (typically IoT or home automation), it can be successfully used. It provides the concept of interconnecting function blocks (in Node-RED terminology, these are nodes) into flows, which corresponds to composites in DEVS. These can be organized hierarchically. When transforming DEVS into Node-RED, it is practically only necessary to overcome a small problem. The Node-RED node has only 0 or 1 input. Nevertheless, the specific meaning of the input data can be distinguished by the topic specification in the incoming message structure (the message object always contains the topic and payload slots). For the systematic transformation of DEVS into Node-RED, it is necessary to express the existence of input ports of the component by a corresponding modification of the message topic. In our case we solve this by nodes of type change (in Figures 10, 11, and 12 they are colored yellow). Such a node sets msg.topic to the corresponding DEVS port name. In a case when DEVS does not define port names explicitly (in our case, the *or* component in Figure 8), generic names like x1, x2 are used (see Figure 11).

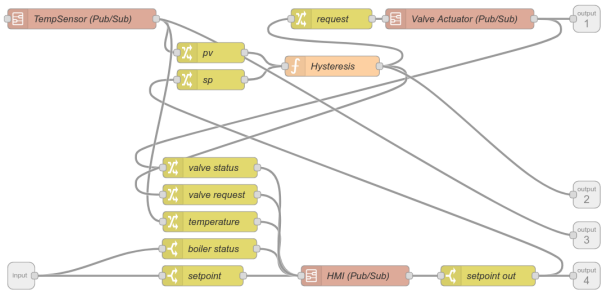

Figure 12. Node-RED model of zone controller.

According to this information, the next node then knows the port name and can handle the message adequately. Node-RED flows in Figures 10, 11, 12 correspond to DEVS models in Figures 7, 8, 9.

C. IEC 61499 implementation

Transformation into IEC 61499 also requires overcoming a slight difference compared to DEVS. IEC 61499 distinguishes between data and events due to the optimal implementation of complex real-time control applications. If data appears on the input port, it does not mean that it should be processed immediately. For the input data to be processed, the corresponding input event has first to be accepted. Which data is processed in which input event is specified by the interface. Figure 13 shows the interface of the Central Heating block type. The jumpers between event and data inputs specify which data is processed at which events. For the purpose of systematic transition from DEVS to IEC 61499, it makes sense for each data input or output to specify its own event.

 ISBN: 978-1-61208-894-5

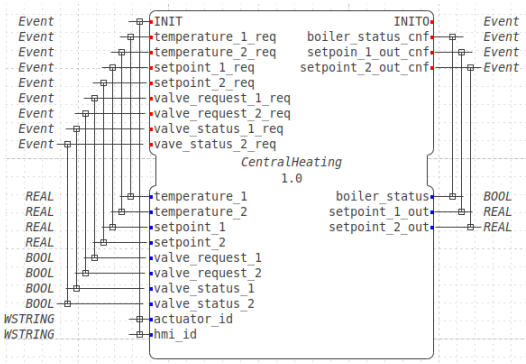

Figure 13. Interface of Central controller. To be DEVS-compatible, each data input and output has its associated event.

Another difference of IEC 61499 from DEVS is that data from multiple sources cannot be sent to single data input. However, this can be solved by including a special component that forwards any of its inputs to a single output. In our example, we do not have such a situation, but in general, it is necessary to count with it.

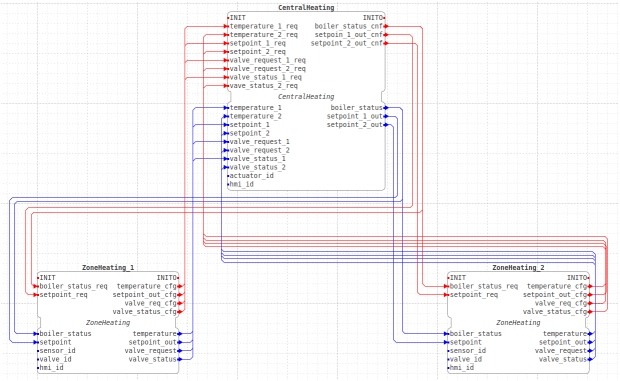

Figure 14. IEC 61499 model of the system.

Figure 14 shows the system model according to IEC 61499 (modeled in 4diac development environment), corresponding to the DEVS model in Figure 7. The other models would be derived from DEVS analogously.

VI. Platform Specific Model – Distributed Version

The entire control application, comprising two zone controllers and one central, can be easily deployed to a single computer or Programmable Logical Controller (PLC), equipped with a runtime environment for PowerDEVS, Node-RED, or IEC 61499. Now let's look at the distributed implementation of the control application. IEC 61499 supports distributed deployment natively. In DEVS and Node-RED, we can do it analogously. We will demonstrate the principle using the IEC 61499 approach.

The first step is a model of a distributed computing environment, see Figure 15. In our case, there are two networked Raspberry Pi computers. The second step is to map the components of the control application to the compute nodes. In Figure 16, it is visualized by different colors of the components

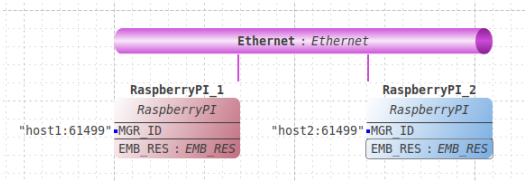

Figure 15. IEC 61499 distributed system model. One RPi hosts Central controller and Zone 1 controller, The second one hosts Zone 2 controller.

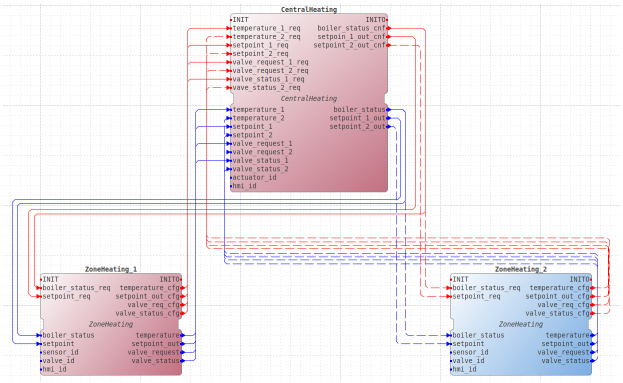

Figure 16. IEC 61499 model of the system - distributed version. Different component colors mean that they are mapped to different nodes of the distributed system. Dotted lines model network connections between parts of the system.

(the colors correspond to the colors of the computational nodes in Figure 15). The third step is to ensure communication between the distributed components using a suitable network protocol. These are the dashed links between the components in Figure 16. The application running on the first or second node must be equipped with communication components for each link that leads out of the node, respectively inside the node, see Figures 17 and 18. In addition, it is also necessary to ensure the initialization of components at the start of the platform (see the component E_START). The selection of

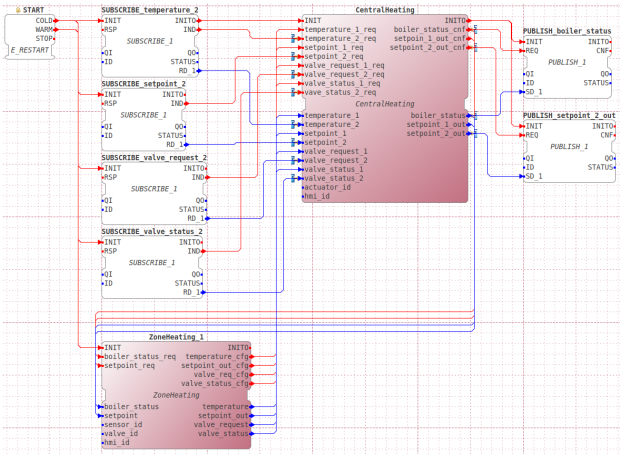

Figure 17. Central controller and Zone 1 controller with an interface to the Zone 2 controller is deployed on Rasberry Pi 1

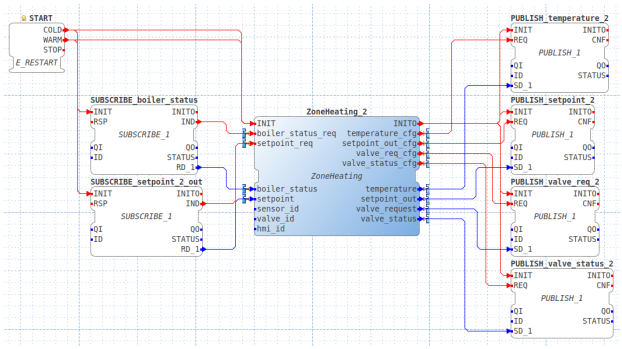

Figure 18. Zone 2 controller with and interface to central controller is deployed on Raspberry Pi 2.

communication components and the way of initialization of components can be standardized for the considered application domain. In our case, we use only the MQTT protocol, message topics are unambiguously derived from component and port names, and all components can be initialized concurrently.

VII. Conclusion

This paper has shown by example the transition from a conceptual model through a platform-independent model to platform-specific models in several environments usable for control applications. The initial modeling tool was the UML language, which is followed by the DEVS formalism. We understand DEVS as an essential tool and concept for modeling and implementing control systems in our approach. We demonstrated the transition from DEVS to three implementation environments. Each of them has specifics regarding distribution, real-time responses of runtime availability for given hardware (e.g., PLC, soft PLC, or PC), and semantics of block diagrams, flows, etc. Specifically, PowerDEVS is suited for time critical regulatory control, but it is also applicable in another way (like in our example). On the other hand, Node-RED is designed to typically run on an IoT gateway or in the cloud, not on a PLC. Finally, IEC 61499 is applicable for PLC as well as for higher levels of control. In addition, it has the means for distributed application deployment. Besides the implementation environments we dealt with, there is possible to consider any other modeling and programming means based on hierarchically organized and interconnected function blocks with semantics similar to DEVS.

An interesting feature of our methodology is the continuity of the DEVS model in all considered implementation environments. The transformation of the DEVS model into the target environment is based on relatively simple rules, and the original structured DEVS model is retained. Because DEVS is at a higher level of abstraction than, for example, IEC 61499, where data and event ports are distinguished, modeling is easier. This more direct modeling is then at the cost of slightly less efficient implementation (the component in DEVS always reacts to any change in any input port). However, it is still usable for hard real-time in the same way as, e.g., PowerDEVS. At the same time, it allows transformation to Node-RED (unless hard real-time behavior is required).

Unlike other related works, we do not deal with the level of atomic blocks. We do not generate them from their specification, but we expect a standard library (concerning the application domain) of atomic function blocks in all considered environments. However, to provide the necessary function block libraries in target environments, their automatic generation from behavioral specifications can be considered using activity diagrams, statecharts, etc., similar to some of the related work mentioned in Section II.

Acknowledgment

This work has been supported by the internal BUT project FIT-S-20-6427.

References

[1] Eclipse. (2021) 4diac documentation. online. [Online]. Available: https://www.eclipse.org/4diac/. [Retrieved: 08, 2021]

[2] OpenJS Foundation, “Node-red,” online, 2021. [Online]. Available: https://nodered.org. [Retrieved: 08, 2021]

[3] R. Manione, “A full model-based design environment for the development of cyber physical systems,” *Design*, vol. 3, no. 1, pp. 1–30, 2019.

[4] F. Ciccozzi, I. Malavolta, and B. Selic, “Execution of uml models: a systematic review of research and practice,” *Software & Systems Modeling*, vol. 18, no. 3, pp. 2313–2360, 2018.

[5] B. Zeigler, T. Kim, and H. Praehofer, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, Inc., London, 2000.

[6] X. Hu and B. Zeigler, “Model continuity in the design of dynamic distributed real-time systems,” *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 35, no. 6, pp. 867–878, 2005.

[7] E. P. Marcosig, J. I. Giribet, and R. Castro, “Devs-over-ros (dover): A framework for simulation-driven embedded control of robotic systems based on model continuity,” in *2018 Winter Simulation Conference (WSC)*. IEEE, 2018.

[8] F. Bergero and E. Kofman, “Powerdevs: A tool for hybrid system modeling and real-time simulation,” *Simulation*, vol. 87, pp. 113–132, 01 2011.

[9] T. Hussain and G. Frey, “UML-based Development Process for IEC 61499 with Automatic Test-case Generation,” in *IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2010.

[10] C. A. Garcia, E. X. Castellanos, C. Rosero, and Carlos, “Designing Automation Distributed Systems Based on IEC-61499 and UML,” in *5th International Conference in Software Engineering Research and Innovation (CONISOFT)*. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8337936

[11] I. A. Batchkova, Y. A. Belev, and D. L. Tzakova, “IEC 61499 Based Control of Cyber-Physical Systems,” *Industry 4.0*, vol. 5, no. 1, pp. 10–13, November 2020.

[12] S. Panjaitan and G. Frey, “Functional Design for IEC 61499 Distributed Control Systems using UML Activity Diagrams,” in *Proceedings of the 2005 International Conference on Instrumentation, Communications and Information Technology ICICI 2005*, 2005, pp. 64–70.

[13] H. T. Park, K. Y. Seong, S. Dangol, G. N. Wang, and S. C. Park, “An Approach to Obtain a PLC Program from a DEVS Model,” in *In Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics - RA*. [Online]. Available: https://www.scitepress.org/papers/2008/14924/14924.pdf

[14] A. Gonzalez, C. Luna, M. Daniele, R. Cuello, and M. Perez, “Towards an automatic model transformation mechanism from UML state machines to DEVS models,” *CLEI Electron. J.*, vol. 18, no. 2, 2015. [Online]. Available: https://doi.org/10.19153/cleiej.18.2.3

[15] G. D. Kapos, V. Dalakas, A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos, “Model-based system engineering using SysML: Deriving executable simulation models with QVT,” in *IEEE International Systems Conference Proceedings*, 2014.

[16] M. Moallemi and G. A. Wainer, “Modeling and simulation-driven development of embedded real-time systems,” *Simulation Modeling, Practice and Theory. Elsevier*, vol. 38, pp. 115–131, November 2013.

 ISBN: 978-1-61208-894-5