Requirements for a dynamic interface model of IEC 61499 Function Blocks

Bianca Wiesmayr, Alois Zoitl, *Member, IEEE*
LIT CPS Lab, Johannes Kepler University Linz
Linz, Austria
{bianca.wiesmayr, alois.zoitl}@jku.at

***Abstract*—Component-based software engineering has emerged as a principle of software design to facilitate reuse and improve the software quality. This principle is supported by the domain-specific language IEC 61499, where Function Blocks are fully encapsulated software components. For a Function Block definition, a static interface description and an internal implementation are required. Service sequences describe the event flow at a component interface and are an optional dynamic interface model in IEC 61499. In general, dynamic interface models are a powerful tool for various use cases, yet service sequences are rarely used in practice due to their low expressiveness. Therefore, we identify the domain-specific requirements for a comprehensive dynamic interface model and use them for our analysis of service sequences, where several issues are identified.**

***Index Terms*—IEC 61499, behavior modeling, service sequence**

I. Introduction

Modern distributed control systems involve a large amount of automation software. Additionally, requirements such as standardization and interoperability of systems need to be fulfilled. Engineers increasingly aim at modularizing hard- and software as a strategy to tackle the growing software complexity [1]. Component-based Design (CBD) is widely applied in conventional software engineering [2] and is also realized in the industrial standard IEC 61499 [3].

Modular and encapsulated components are essential to realize a CBD that fosters reuse. It has several advantages: The development effort for software can be reduced when reusing well-defined components that are tested and documented. This allows cost reductions while improving the software quality [2]. Maintainability of software is essential in industrial automation due to the long lifecycles of plants [4]. If an appropriate modularization is considered in the software design, the maintainability is greatly improved by CBD because components can be replaced independent of the rest of the software [2].

Model-driven software engineering (MDSE) has emerged as a strategy of reducing the complexity of a real-world problem by abstraction, similar to models used in physics and engineering. Software models are seen as the primary development artifact and transformations from a source model to a target model are an integral aspect in MDSE. Modeling is supported by modeling tools which support import and export of models, ensure correct syntax and semantics, and provide code generation and/or model transformation [5]. IEC 61499 is an example for a domain-specific modeling language (DSML), where the components are Function Blocks (FBs). The MDSE principle is realized by generating code that can run on control devices. Besides the application model, which is comprised of a composition of FBs, also the system configuration is modeled. A mapping between the application and the devices guarantees a sufficient abstraction from the hardware level [6].

To benefit from CBD, developers need to regularly update and maintain software components. Furthermore, a comprehensive documentation is essential for developers in order to understand a component's behavior and identify compatible components [2]. This documentation is particularly relevant for IEC 61499 because the DSML is incomplete, i.e., additional languages are needed to develop a software system. Algorithms, which may be specified in any low-level programming language, are linked to the DSML and IDEs typically provide tool support for specifying algorithms based on IEC 61131-3. Other challenges, such as the versioning of FBs, need to be addressed by developing advanced tool support.

In this paper, we analyze service sequences, which are dynamic interface models in IEC 61499. Such models can enhance a static port declaration. In the next section, we analyze related work. The scope of service sequences is discussed in Section III. Advantages and applications of dynamic interface models are outlined in Section IV.A. Based on these use cases, requirements are derived in Section IV.B. Finally, service sequences are analyzed with regard to these requirements in Section V. The paper is concluded in Section VI by presenting the next steps of our research.

II. Related Work

The information provided in behavior models can be used for validating and verifying system models. Simple compatibility rules for components were derived in a hierarchical architecture by using service sequences to describe behavioral event contracts for adapters [7]. The benefit of modeling can be further increased when a model is used in many steps of the development process. Service sequences can be used for monitoring the behavior of an application during runtime [8].

To extend the modeling capabilities for control systems, IEC 61499 has been combined with the general-purpose models of UML [9]–[11]. UML modeling was used as a tool for planning the high-level architecture [9]. An extended methodology for using UML in conjunction with IEC 61499 has been proposed by [10]. The applied model transformations are usually not reversible.

Numerous languages for component models are described in the literature, some of which are targeted at a specific application domain. A thorough overview and classification can be found in [3]: For instance, languages can be distinguished based on the proposed interface specification for components. IEC 61499 currently focuses on definitions of compatible data types, while other languages also allow behavioral contracts.

III. Behavior Modeling in IEC 61499

IEC 61499 [12] standardizes several kinds of FBs, all of which have the same static interface declaration that consists of input and output ports for the data and event flow, including their respective event/data types. These kinds of FBs are differentiated based on how their behavior is specified:

- Basic FBs have a state diagram that controls the execution of actions (Execution Control Chart, ECC). Actions consist of an algorithm and/or an output event.
- Service Interface FBs (SIFBs) are typically used for hardware access. They implement functionality that cannot be modeled within IEC 61499. Such FBs are programmed for example in C++.
- Composite FBs and subapplications are hierarchical language elements and their behavior depends on the enclosed FB network.

The standard additionally defines that each FB may have any number of service sequences, which are sequences of service primitives. Service primitives show a possible scenario of the event flow between the internals of a FB and the connected FB network [12]. In contrast to the ECC, they are independent of a concrete implementation of the FB behavior because they model the externally observable behavior of a component. Such models are often termed “dynamic interface descriptions” [3] as they enhance the static port specification.

The basic structure of a service sequence is shown in Fig. 1. The sequence has a name which is displayed at its top and can describe the modeled scenario. On the left side, the perspective of the FB network (i.e., the application using an FB instance) is displayed. On the right side, information about the FB internals are provided. After the event EI arrives at an input port, it triggers an execution within the FB (processEI). An output event EO is sent upon completion of processEI.

Service sequences may also be defined for adapter interfaces, which are typed ports that consist of data and events. For adapters, the communication between plug and socket is modeled. Additionally, the standard defines a slightly adapted notation for SIFBs, for which several port names are reserved. For example, the boolean input qualifier named QI is associated with the INIT event. The service is supposed to be initialized when INIT occurs and QI is true. In contrast, the service is terminated when QI is false. These variations can be illustrated in the service sequence as INIT+ and INIT-.

Unfortunately, the expressivity of service sequences is limited as explained in [7]. Additional features and possibly even additional models will be required. In the following section, we analyze the domain requirements for a possible extension of the standard IEC 61499.

IV. Requirements Analysis

We first consider the applications of a dynamic interface model to then derive requirements.

A. Use Cases of Dynamic Interface Descriptions

Based on the state of the art and our analysis we identified the following use cases (UCs) throughout the development process:

1) *Developing architectures:* The design process starts with capturing requirements and describing the dynamic behavior, for which models such as use case diagrams and sequence diagrams are used in MDSE [10]. Models aid in the requirements engineering process and, as a communication tool, contribute to efficient collaboration within developers [13]. The application architecture and design is planned before the actual implementation [14]. So, not all FB implementations may be available at the start of the development process, but dynamic interface descriptions modeling the expected behavior can allow first compatibility checks.
2) *Component documentation:* [2] A comprehensive documentation is particularly beneficial in IEC 61499 because any language can be used within an FB, so users of this FB may not understand the algorithms. They may even lack access to them due to copyright protection by the vendor of a software component.
3) *Abstraction:* A high quality interface definition detaches the component specification from its implementation details. This abstraction is particularly relevant for complex software systems that have several hierarchical levels. For hierarchical components, a dynamic interface description provides a better overview to the user, while currently, the behavior is only implicitly modeled as a composition of FBs. Users need to derive the functionality of the hierarchical component from its enclosed FB network and are therefore required to understand the behavior of subcomponents.

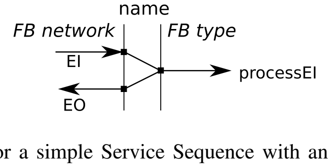

Fig. 1. Example for a simple Service Sequence with an input event EI and an output event EO.

4) *Testing:* After implementing a component, it can be tested against its specification via unit-testing. In test-driven development, the tests are specified before the actual implementation. For example, a formal model can be used as a specification [15]. Additionally, interface models can be used to monitor the behavior during runtime [8] (integration test).
5) *Validation and Verification:* Tools can support the developer by providing automated checks between the implementation and the interface specification. Furthermore, the compatibility of components can be analyzed automatically. Such checks may detect incompatibilities already at design time. Early error detection can even facilitate development and reduce costs [13]. In some disciplines, verification is essential to fulfill strict safety requirements [14].
6) *Refactoring:* Refactoring aims at improving or updating the implementation while preserving the behavior that is observable at the component interface [1]. An interface specification can prevent unintendedly altering that behavior.

B. Derived Requirements

Developers of control software are typically not experts in software engineering [1], so the usability of a model is a core focus. Several notations are commonly used to evaluate the usability of languages, such as the cognitive dimensions [16]. These parameters can guide DSL engineers and tool developers. A high usability of the DSML can also improve the adoption of IEC 61499. Based on the use cases UC1 to UC6 that were presented in the previous section, we can identify further requirements:

R1) *The model captures the domain information.*

Real-time systems such as automated production systems typically react to external events (reactive systems). IEC 61499 supports both event and data flows, the behavior model has to capture both to fully reflect the FB behavior. Besides the typical simple data types, also datatypes representing bit-sequences are standardized. Additionally, composite datatypes have to be considered, which are called Structured Types in IEC 61499. For modeling data flow, the WITH-relation has to be taken into account because data are only updated when an associated event occurs. Some applications require specifying a range of data values. This will allow modeling if-conditions where the behavior of a FB depends on data inputs.

The following features are required for efficiently modeling the event flow at an interface: (1) For FBs with multiple event inputs, the order of event occurences can be specified. (2) It can be indicated that certain events must occur but their order is arbitrary. For example, a synchronization block of several paths has to send an output event as soon as all input events have been received. (3) A multiplicator of the event occurence can be specified. Sometimes, an event has to be received for a specified number of times. For instance, a filter or a counter may only send an output event after every n-th input event. It is furthermore desired that event types can be represented, although there is currently only limited tool support for them.

Finally, it must be possible to model real-time constraints (UC4) because the system behavior often relies on specifying a worst-case execution time [14] or simply timer functions.

R2) *Completeness: Developers can create both an incomplete or a complete model of the system.*

During the first phases of the development process (UC1), part of the specification may still be missing. Possible consistency checks may however already be possible at this stage, so developers could benefit from an early version of the model. This requirement is also reflected in the cognitive dimensions as “progressive evaluation” and “premature commitment” [16]. The former illustrates the benefit of early feedback on a model, while the latter describes possible barriers in the development progress if all information is required before designing a model. An incomplete modeling language may however encourage developers to use a “secondary notation” [16], for example by adding comments to a model. These comments are usually not interpreted by tools and can lead to inconsistencies and maintenance issues. Therefore, complete models must be supported for a full specification of the component (UC5). The behavior of a component has to be describable in a finite number of models [7].

R3) *The notation is well readable for domain experts.*

Formal descriptions such as temporal logic are frequently used in computer science, but such notations are not typically understood by the domain experts [1]. As users should be able to understand and use a component solely based on the interface specification (UC2), an intuitive notation is required. A modeling approach can avoid misinterpretations compared to a purely textual documentation. In the automation domain, graphical languages are common (e.g., ECC, Sequential Function Chart [17], GRAFCET [18]). Please note that this requirement only concerns the concrete syntax, i.e., the representation of the model that is shown to the user.

R4) *A formal description of the model can be derived.*

In addition to the human-readable representation, the model should be described formally. This is essential for generating test cases (UC4) or automated checks (UC5). A chosen model should already have at least one formal representation to ensure using a well-established and extensible notation.

R5) *The same model is usable for all kinds of FBs.*

The port declaration of all FBs (and even subapplications) is equivalent. If also the dynamic interface model is independent of the FB kind, the interface description can therefore be fully detached from the implementation. This allows a generic interface declaration before deciding on the block type (UC1). Furthermore, developers need to refactor code (UC6). For example, a basic FB of high complexity may be converted into a hierarchical component, while its interface description should remain unchanged.

R6) *Extensive tool support is provided.*

The benefits of DSMLs depend also on the provided tool support [5]. Because the language is tailored to a specific domain, also the tooling can be adapted to the needs of automation engineers [14]. Additional views for the application model, such as the intended dynamic interface model, bear the risk of inconsistencies. IDEs can reveal such inconsistencies and thus greatly reduce the error proneness of the model. Additionally, tool support has to be implemented so that the Use Cases 4 to 6 become feasible.

V. Current status of service sequences

Service sequences already fulfil some of the requirements.

Concerning R1, service sequences cannot model data values nor timing information, but they fulfil most of the needs for representing the event flow. Either an order of events is enforced or a symbol (~) indicates an arbitrary order. The number of event occurrences can however only be specified by repeating this event. Regarding R2, service sequences do not provide complete models in all cases. As an example, a loop within the ECC may require an infinite number of service sequences to provide a full representation of the FB behavior [7]. If-conditions cannot be modeled due to the lack of data values. For example, a control block may control the movement of a conveyor belt. Based on a sensor input, the conveyor belt has to move either forwards or backwards. We'd need the possibility to specify the range of the sensor input for a certain service sequence. While this is already possible as part of the sequence name, this is considered an unwanted "secondary notation" [16]. The text cannot be interpreted by tools leading to undetected errors. It is expected that R3 is fulfilled for the current notation as only few language rules apply. Scenario-based models are well-suited for communicating ideas and documentation [19]. Service sequences can furthermore be used for all types of FBs (R5) including subapplications, although the standard particularly recommends them as a tool for modeling SIFBs. Unfortunately, the tool support (R6) for service sequences remains restricted. For example, the standard allows modeling the WITH-relation not only in the static interface declaration, but also in the service sequences. To our knowledge, no IDE reveals these potential inconsistencies. Current IDEs furthermore provide only limited support for the UCs 4-6.

VI. Conclusion and Future Work

In a domain representation, users can specify the behavior of a system by defining expected and forbidden traces of its execution. Based on this, a formal representation is derived for verification purposes [13]. To check for semantic compatibility between components, their expected behavior is usually formalized as protocol state machines, preconditions, or postconditions [14].

In this paper, we showed that service sequences currently do not fulfil all requirements that are relevant for industrial automation. Hence, we will next propose an extended version of service sequences which fulfil the requirements presented in this paper. Such extensions could for instance be based on the development of sequence charts, as a vast number of publications considering notations (cf. [20]) and formal techniques (cf. [13]) are available in the literature. Message Sequence Charts include loops, alternatives, and hierarchical elements [20]. Live Sequence Charts have an additional feature for specifying forbidden scenarios [19]. It was attempted to synthesize statecharts as intra-object specifications from these sequence charts [19]. Such model transformations may reduce the modeling effort.

After identifying possible extensions and a corresponding formal description, we will focus on providing tool support for this new notation to increasingly benefit from its potential. Extensions to the open-source tool Eclipse 4diac [21] are envisaged. Then, a study based on the cognitive dimensions [16] may be required to ensure the usability of our approach by domain experts.

References

[1] B. Vogel-Heuser and A. Sarda-Espinosa, "Current status of software development in industrial practice: Key results of a large-scale questionnaire," in *IEEE Int. Conf. on Industrial Informatics (INDIN)*, pp. 595–600, 2017.

[2] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. d. M. Silveira Neto, Y. C. Cavalcanti, and S. R. d. L. Meira, "Twenty-eight years of component-based software engineering," *Journal of Systems and Software*, vol. 111, pp. 128–148, 2016.

[3] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Trans. on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[4] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[5] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2nd ed., 2017.

[6] A. Zoitl and R. W. Lewis, *Modelling control systems using IEC 61499*, vol. 95 of *IET Control engineering series*. London: IET, 2nd ed., 2014.

[7] H. Prähofer and A. Zoitl, "Verification of hierarchical IEC 61499 component systems with behavioral event contracts," in *IEEE Int. Conf. on Industrial Informatics (INDIN)*, pp. 578–585, 2013.

[8] M. Wenger, A. Zoitl, and J. O. Blech, "Behavioral type-based monitoring for IEC 61499," in *IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–8, 2015.

[9] S. Panjaitan and G. Frey, "Combination of UML modeling and the IEC 61499 function block concept for the development of distributed automation systems," in *IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, pp. 766–773, 2006.

[10] C. Tranoris and K. Thramboulidis, "Integrating UML and the function block concept for the development of distributed control applications," in *IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, vol. 2, pp. 87–94 vol.2, 2003.

[11] T. Hussain and G. Frey, "UML-based development process for IEC 61499 with automatic test-case generation," in *IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, pp. 1277–1284, 2006.

[12] IEC, "IEC 61499-1, function blocks - part 1: architecture," 2012.

[13] S. Gabmeyer, P. Kaufmann, M. Seidl, M. Gogolla, and G. Kappel, "A feature-based classification of formal verification techniques for software models," *Software & Systems Modeling*, vol. 18, no. 1, pp. 473–498, 2019.

[14] M. Völter, *DSL engineering: Designing, implementing and using domain-specific languages*. Lexington, KY: CreateSpace Independent Publishing Platform, 2010-2013.

[15] R. Hametner, I. Hegny, and A. Zoitl, "A unit-test framework for event-driven control components modeled in IEC 61499," in *IEEE Conf. on Emerging Technology and Factory Automation (ETFA)*, IEEE, 2014.

[16] A. Blackwell and T. Green, "Notational systems—the cognitive dimensions of notations framework," in *HCI Models, Theories, and Frameworks*, pp. 103–133, Elsevier, 2003.

[17] IEC, "IEC 61131 - programmable controllers, part 3: Programming languages," 2013.

[18] IEC, "Specification language GRAFCET for sequential function charts: IEC 60848," 2013.

[19] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods (Ehrig Festschrift)*, vol. 3393, pp. 309–324, Springer-Verlag, 2005.

[20] I. H. Krüger, *Distributed System Design with Message Sequence Charts*. Dissertation, Technische Universität München, München, 2000.

[21] Eclipse 4diac, "Eclipse 4diac - the open source environment for distributed industrial automation and control systems," 2019.