JOT
JOURNAL OF
OBJECT TECHNOLOGY

An Interpreter-Based Execution Framework for IEC 61499 Function Block Networks

Bianca Wiesmayr[*,†,‡], Antonio Garmendia[§], Alois Zoitl[**], and Manuel Wimmer[*]
[*]Institute of Business Informatics - Software Engineering, Johannes Kepler University Linz, Austria
[†]Linz Institute of Technology (LIT), Johannes Kepler University Linz, Austria
[‡]Institute for Software Engineering and Programming Languages, Ulm University, Germany
[§]Department of Computer Science & Statistics, King Juan Carlos University, Spain
[**]LIT CPS Lab, CDL VaSiCS, Johannes Kepler University Linz, Austria

ABSTRACT Control software of cyber-physical production systems can be modeled using the domain-specific language defined in the IEC 61499 standard. The event-based execution model of this language accurately reflects the behavior of distributed control systems and its block-based diagrams facilitate component orientation. However, when applying the standard to larger-scale automation systems, engineers require sophisticated methods and tools. In particular, current testing methods rely on executing control software after its deployment to dedicated devices or runtime environments. This typically requires a complex setup involving control hardware, software, and communication infrastructure, which is time-consuming and error-prone. In contrast, the aim of this paper is to enable early feedback during the development of control software by interpreting the hardware-independent Application model. Our interpreter-based execution framework for IEC 61499 Function Block (FB) networks directly executes the software model, records execution traces, and visualizes these traces, e.g., for testing purposes. We have evaluated the functional correctness of our approach with respect to the execution semantics of the IEC 61499 execution environment 4diac FORTE by using a compliance test suite. Although the interpreter does not mimic the real-time behavior of the cyber-physical production systems, developers receive early feedback on the event-based execution logic. According to our scalability test, interpreting medium-sized control applications (a few thousand executed FB instances) is fast enough to provide control engineers with rapid feedback on the implemented behavior. We also used our interpreter and its accompanying tools for analyzing an existing IEC 61499 Application, which showed that the interpreter can assist in the development process. Overall, the evaluation shows that Application models can be efficiently interpreted for providing early validation feedback to engineers before moving to hardware- and communication-specific tests.

KEYWORDS Control software, Model-driven engineering, Model interpreters, Development tools.

1. Introduction

The requirements of modern production systems with respect to flexibility and adaptability pose high demands on the engineering of control systems. Control software reads data from sensors and aims to influence its environment. For instance, a motor may be controlled depending on the measured inflow of materials. This tight coupling often makes software testing and debugging difficult, especially at the conceptual stage without access to any hardware. Control software that is executed on Programmable Logic Controllers (PLCs) (Lyu & Brennan 2021) is usually developed based on industrial standards, i.e., IEC 61131-3 (IEC 2025) or IEC 61499 (IEC TC65/WG6 2012)–both representing executable languages. IEC 61499 is a domain-specific modeling language for distributed control software, which is supported

JOT reference format:
Bianca Wiesmayr, Antonio Garmendia, Alois Zoitl, and Manuel Wimmer. *An Interpreter-Based Execution Framework for IEC 61499 Function Block Networks*. Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.3.a19

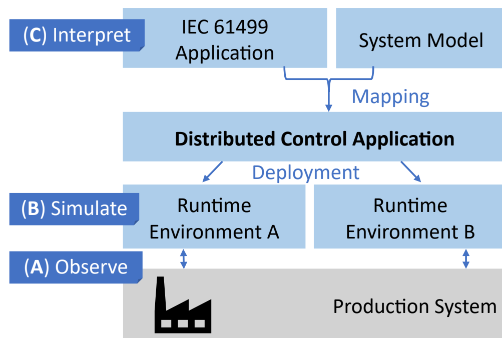


Figure 1 Mechanisms for testing and evaluating control software: (**A**) Observing interactions with a real machine, (**B**) execution in a control device simulator, (**C**) interpreting platform-independent control software in the tool.

by development environments from various vendors, cf. the overview in (Lyu & Brennan 2021). The programming models of PLCs differ from those for general-purpose software as they are adapted to the requirements of automation. For instance, the standards provide specific functionality to ensure reliable control systems and to enable reactions to external events (Sehr et al. 2021). Due to these characteristics, domain-specific tools and software engineering methods are needed. Event-based execution models (as defined by IEC 61499) and virtual prototypes for testing can help handling system complexity (Sehr et al. 2021), as well as model-based engineering to raise the abstraction level compared to programming directly in PLC programming languages (Vyatkin 2013).

IEC 61499 separates the implemented functionality from the hardware infrastructure (cf. top of Figure 1) (Zoitl & Vyatkin 2009). Hence, engineering tools can provide means for platform-independent testing and simulation, allowing for an *evaluation of event-based functionality in the early stages of development* (Zoitl & Lewis 2014, p. 29). The Application model implements the logical functionality of control software in a network of Function Blocks (FBs) and can be distributed across devices that are represented in the System model. It is independent of any vendor-specific technologies (Wenger et al. 2011). This so-called "mapping" creates the platform-specific model of the distributed control software (Zoitl & Vyatkin 2009), which is extended with hardware-specific functions (e.g., for communication) (Wenger et al. 2011), and thus, cannot be ported between platforms (Vyatkin 2011). This means that software tailored to a runtime environment (RTE) cannot be easily executed in another one. Moreover, testing the platform-neutral IEC 61499 Application is currently not supported in IEC 61499 environments. This means that developers need to introduce hardware dependencies early in the development process to assess the correct functionality of their software. Our goal is to design and realize an infrastructure for evaluating the Application model.

Figure 1 illustrates the different alternatives for evaluating control software. The available technologies support (**A**) observing data and event flows between the production system and the control software, as well as (**B**) simulating the control software within a specific runtime environment and observing the virtual inputs and outputs (Cabral & Zoitl 2025). Software simulation does not require access to real hardware, thus, enabling continuous integration tests and facilitating the test of fault scenarios. Simulation environments should be integrated with testing and verification tools through standardized models (Vyatkin 2013) (e.g., defining simulation scenarios as models). Evaluation mechanisms **A** and **B** include real-time execution and communication between RTEs. However, software execution in an RTE requires prior code generation and deployment to the RTE based on the system model, leading to a long round-trip time for developing Application models. Furthermore, any testing/debugging infrastructure is tailored to a specific RTE, rather than enabling platform-neutral engineering. To validate the Application model itself, a (**C**) interpreter for the IEC 61499 Application model can bring additional benefits. A **model simulation** has increased interaction possibilities compared to execution in an RTE because developers can flexibly adjust the internal states of software components or values of variables. Mechanisms for analyzing partial models during the development can further reduce the development effort, especially for novice engineers. In event-based FB networks, only FBs that are affected by the initial trigger event are executed. Whereas deployment to an RTE requires syntactically correct models, only the actually executed parts need to be complete for successful interpretation. Furthermore, access to inputs and outputs, communication mechanisms, or fault detection may not be implemented in the early development phases and can be abstracted more easily than in simulation. Hence, this paper tackles the challenge of *interpreting the IEC 61499 Application model, without requiring information about the system configuration*. This approach, together with *infrastructure for testing control software*, is our core contribution. Due to semantic ambiguities of the standard (Wiesmayr et al. 2023), each IEC 61499 RTE implements a specific execution model. This paper demonstrates the approach based on the execution strategy implemented in the RTE 4diac FORTE, a part of the open-source environment Eclipse 4diac (Eclipse 4diac 2025).

The remainder of this paper is organized as follows. Model interpretation as a platform-independent execution mechanism has already been successfully used for other modeling languages (Section 2) and will be applied to the event-triggered execution of IEC 61499 Function Block (FB) networks. The language elements involved (including syntax and semantics) are described on the basis of a running example (Section 3). An interpreter for software models (Section 4) allows for direct feedback in the development tool without requiring specific hardware or even an RTE. The provided framework (Section 5) includes (i) an interpreter implementation of the operational semantics for FB networks, including a mechanism to save execution traces in a platform-independent format; (ii) a graphical visualization of the execution traces for inspection; and (iii) an automated comparison of different execution traces. We cover the IEC 61499 language elements of FB networks (excluding hierarchical structures) and compare the results of our execution framework with those obtained from 4diac FORTE based on a test suite (Section 6.1) and an application example (Section 6.3) to assess the correctness and completeness of our interpreter. We also

measured its performance to evaluate the scalability for the simulation of control software (Section 6.2), and discuss threats to validity in Section 6.4. Finally, we conclude the paper with an outlook on future work (Section 7).

2. Related Work

2.1. Interpreting Models

Model-driven engineering often specifies modeling languages as meta-models (Brambilla et al. 2017; Wachsmuth 2007). The standard IEC 61499 includes a proposal for a meta-model of the main language elements (IEC TC65/WG6 2012). A realization of this meta-model using the Eclipse Modeling Framework (EMF) is available as part of Eclipse 4diac (Eclipse 4diac 2025). In contrast to code generation from models, a model interpreter is a generic execution engine that runs the model itself. This approach allows for rapid feedback during development, facilitates debugging, and increases the level of abstraction during the engineering of software models (Brambilla et al. 2017). Generally, two approaches are considered for model interpretation, i.e., semantic description and translational approaches (Wachsmuth 2007). While the former relies on extending the language for execution concerns, the latter is translating a given model expressed in a source language into an executable model expressed in a target language. We use a semantic description approach as we extend the IEC 61499 meta-model for describing the operational semantics of the IEC 61499 language elements.

Similar approaches to ours were standardized by the Object Management Group (OMG) by proposing Foundational UML (fUML) (Object Management Group 2021) and the Action Language for fUML (Alf) (Object Management Group 2017). fUML is a language for the execution of a subset of UML (Object Management Group 2015) that covers the core elements and their specific semantics. In order to complement fUML, the OMG specified Alf to provide a textual language to represent the executable behavior. Furthermore, the precise semantics for state machines extends fUML and covers UML Statecharts (Elekes et al. 2023). Recently, formal definitions of the semantics, together with interpreters, have been also proposed, for instance, for domain-specific languages in the domains of safety-critical real-time software (Colaço et al. 2023), industrial control systems (Zhou et al. 2022), embedded systems (Suzuki et al. 2023), and textual control code (Huang et al. 2019). Furthermore, an interpreter for IEC 61499 Function Blocks, i.e., individual software components, has been used to execute state diagrams directly within an IDE (Wiesmayr et al. 2021). This included an environment for unit testing of FBs. However, no support for executing or testing FB networks, which are composed of interacting FBs, is provided.

Unfortunately, the semantics of modeling languages can vary between different artifacts, such as specifications, test suites, and simulators. Although recommendations on language design (Elekes et al. 2023) can improve future language definitions, existing variants often cannot be successfully harmonized. Thus, they need to be managed instead. For state charts, Exelmans et al. (Exelmans et al. 2022) proposed an implementation for an interpreter and a compiler that handles different variants of execution semantics. Their work also included a feature model to illustrate the semantic variability points of state charts. Their goal was a broad coverage of variations, rather than an optimal representation of a specific execution environment. As the interpreter covers a combination of possible variants, it is not restricted to a few existing variants, but evaluates all possible combinations of language features (Exelmans et al. 2022). Using the approach, an existing interpreter that represents the execution semantics of one platform can be extended to cover additional platforms. Our work focuses on IEC 61499-based software execution as realized in 4diac FORTE, the runtime environment provided by Eclipse 4diac (Eclipse 4diac 2025), but our provided implementations will be designed for extensibility to cover additional variants in the future.

2.2. Testing and Debugging Control Software

Dependable systems engineering requires a thorough description of the semantics, especially in industrial automation, where correctness with respect to functionality and timing are essential and difficult to assess manually (Zhou et al. 2022). Verification can also help in the design of reconfigurable systems (Guellouz et al. 2019). However, during the early stages of systems design, testing and debugging features that are directly integrated with the development environment are essential for rapid feedback.

Tests can be executed on a PLC or through simulation. The latter has advantages with respect to controlling parameters such as communication and can reduce costs (Sinha et al. 2019). Model interpretation is not sufficient for testing a control system because the software will interact with physical components in real-time. However, hybrid approaches combine model interpretation during the development with code generation (Brambilla et al. 2017). The generated code can be used for system tests, where the actual inputs and outputs are observed.

Executing tests on the physical equipment is time- and resource-intensive. With the goal of cost reduction in mind, (Ulewicz et al. 2017) presented an approach to estimate the test coverage for system tests of manufacturing equipment. This allowed prioritizing tests that are more likely to help identifying faults. However, they focused on testing the overall system. Such tests are typically performed during the later stages of the development process. In contrast, testing (parts of) an Application model is already feasible during the development, such as component tests for individual FBs. Service sequences were described as a suitable mechanism for specifying FB tests as models (Hametner et al. 2014). Executing these tests requires an execution environment. A test runner can execute them in an RTE and automatically evaluate the results (Hametner et al. 2014), but this is specific to one RTE. The model interpreter for individual FBs can also use service sequences as a test specification, or derive them from an existing implementation (e.g., to evaluate the results of software maintenance) (Wiesmayr et al. 2021). Tests for control software have also been derived from a higher-level specification (Hametner et al. 2013), or automatically generated based on the implementation (Buzhinsky et al. 2015). To the best of our knowledge, model-based testing has not been investigated for networks of FBs, i.e., complete IEC 61499 Application models.

Debugging covers all aspects from observing a failure in the software to isolating and fixing defects in the code or in the model. It can be performed manually or with tool support. For state charts that were integrated into an IEC 61131-3 tool, debugging support at the model level has been provided (Schütz et al. 2014). Also for PLC code written in IEC 61131-3, comprehensive debugging support that involved monitoring and replaying behavior has been proposed (Werner et al. 2020; Prähofer et al. 2010). Replay debugging can be based on execution traces that are recorded from the automation system. In this case, tracked data includes data from sensors, actuators, and internal states (Werner et al. 2020). A major advantage of replay debugging is that execution traces can be recorded with minimal effect on the real-time execution of a PLC program (Prähofer et al. 2010). In IEC 61499, faults are typically observed manually by "watching" the occurrence of events and the values of the data pins during execution. With the aim to reduce this effort, tool support for debugging in an IEC 61499 IDE was recently presented by (Akifev et al. 2023). It also uses recorded execution traces from an RTE (Akifev et al. 2023; Liakh et al. 2022). Our work can form the basis for debugging features, as it provides execution traces that are obtained from an interpreter. By analyzing these traces, the fault detection methods that are available in the literature (Akifev et al. 2023; Cabral & Zoitl 2025) can be applied to our model execution framework. We envision a hybrid engineering method for IEC 61499 FB networks, since the proposed model execution framework is optimized for interpreting the platform-independent Application model during development time, while the final software is executed in an RTE for any real-time execution tests.

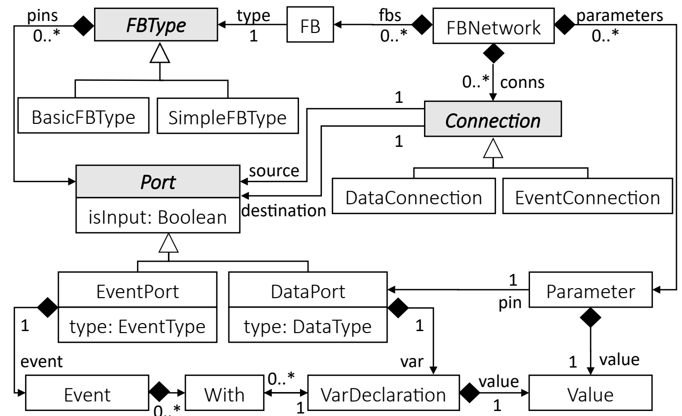


Figure 2 Excerpt of meta-model of IEC 61499 covering the language elements of FB Networks. The main elements are FB instances, connections, and parameters.

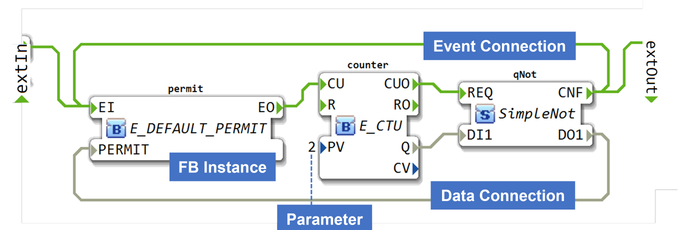


Figure 3 Running example: For-loop counting up to `PV=2`. Any loop content that is connected to extOut (*right*) will be triggered twice. Icon B refers to a `BasicFBType`, icon S to a `SimpleFBType`.

3. Executable Model Elements in IEC 61499

This section outlines background regarding the platform-independent Application model. We focus on elements relevant for the execution of FB networks. The execution semantics of IEC 61499 is defined in the standard using natural language (IEC TC65/WG6 2012). The analysis of the IEC 61499 standard provides the main requirements for developing a model execution framework that is compliant with the standard.

3.1. Language Elements and Running Example

Figure 2 shows an excerpt of the IEC 61499 meta-model. The language elements shown in this meta-model were identified as relevant for the execution of elemental FB networks. Each FB network consists of nodes (FBs) and edges (data and event connections). Additional parameters configure the network (IEC TC65/WG6 2012). As hierarchical elements are not yet supported by the model execution framework, they are omitted in the meta-model. An FB network is the main part of the Application model.

The behavior of an FB network is defined by the interactions of its components but is not explicitly represented in the model. Instead, the actual behavior has to be determined during the execution based on the generated and processed events. After an initial event trigger, further events that control the execution may be generated based on the internal implementation of the executed FBs. The initial trigger of real systems is typically provided based on sensor values or periodic event generators. Events are processed based on a pre-defined execution model. The current execution terminates as soon as no events are left to be processed. The evolution of the values in the for-loop is visualized in Figure 9 (T0-T3 and T5-T7).

We will describe the meta-model elements based on a running example. Figure 3 depicts a model representing a *for-loop* that counts to two. The loop is triggered from an event source (abstracted as `extIn`) that is connected to the event port `permit.EI`. The actual loop content, i.e., the behavior executed within the loop, has to be attached to the output event port extOut. We will show different loop content variants in the evaluation part of this paper (Figure 10 in Section 6).

3.1.1. Function Block instances

represent certain functionalities. FBs are blocks that consist of an interface definition and an implementation of encapsulated functionality (Zoitl & Lewis 2014). The three instances in the running example are named `permit`, `counter`, and `qNot`. Each FB instance has a *type definition* (`FBType`) that implements the behavior as a state diagram (`BasicFBType`), a textual algorithm (`SimpleFBType`), or even a program in a general-purpose language (`ServiceInterfaceFBType`). The semantics of executing FB types was described in (Wiesmayr et al. 2021). For

instance, the type of FB counter is E_CTU, which has a state diagram that implements the count-up function. The inverter qNot is an instance of a SimpleFBType that contains only a single algorithm, which is activated when an event occurrence is received at port qNot.REQ. Algorithms are usually implemented in the language Structured Text (ST), which is optimized for real-time control systems (IEC 2025). It offers typical programming constructs like statements, conditions/loops, and methods, but no dynamic memory allocation or recursion. Each FB type declares an interface that is comprised of Ports. For instance, FB permit has the two EventPorts permit.EI and permit.EO, each representing an Event. permit.PERMIT is a DataPort representing a VarDeclaration, which has a data type and a Value that conforms to this data type.

3.1.2. Connections

exist between two ports whose type matches. They can be differentiated into DataConnections and EventConnections. The former define the event flow, while the latter forward results from a calculation. Figure 3 shows event connections in green (e.g., between permit.EO and counter.CU) and data connections in grey (e.g., between counter.Q and qNot.DI1).

3.1.3. Parameters

configure the FB network. A Parameter specifies an initial Value of a port, which is used for any calculations unless a value was provided via a connection. In the running example, the final loop value is determined by the parameter value 2 set at counter.PV.

3.2. Execution Semantics of FB Networks in the 4diac FORTE

The standard IEC 61499 does not specify all available variability points in the execution semantics explicitly. This subsection therefore describes the execution semantics realized in 4diac FORTE, which is replicated by the interpreter. The main variability points of the IEC 61499 execution semantics were collected in order to replicate the implementation of 4diac FORTE, and to provide an extensible implementation for integrating further execution semantics. A test suite for evaluating the implemented semantics (Wiesmayr et al. 2023) was used for identifying 4diac FORTE's execution behavior.

We will discuss the execution semantics based on the running example. For this, we assume that an external event, provided by the external connection extIn, arrives at the port permit.EI (a so-called event occurrence EO), where it acts as an instantaneous trigger. Based on this example, we discuss the execution semantics of each meta-model element.

3.2.1. Function Block instances

When an EO arrives at an EventPort of an FB instance, it triggers the execution of the contained FB behavior. Only a single EO may arrive at an FB at a time (IEC TC65/WG6 2012). The executing container has to schedule the events accordingly and provides an event queue to prevent any events from being discarded. In response to this incoming EO, an instance may furthermore issue (0..*) output EOs at (0..*) event ports. Executing the functionality will typically update the values that are part of the output data ports.

For example, FB permit passes an input EO on to the output if the value of permit.PERMIT is true. When an EO arrives at any event port of the FB counter, it issues the corresponding output EO, but additionally updates the values of both data ports. When receiving an EO at counter.CU, the variable counter.CV is increased, followed by an EO issued at counter.CUO. Each FB instance of the running example issues only a single output EO. When running an FB network, the EOs and data values need to be transported between FB instances.

3.2.2. Event connections

When a new EO is created at an output event port, it needs to trigger all connected input ports. A single event port can have multiple incoming connections, which realize an or-logic (a so-called fan-in). The FB is triggered when an EO is delivered via any of these connections. In the case of several outgoing connections, all connected FBs are triggered by a so-called fan-out connection.

For the running example, an event occurring at permit.EI will result in a newly created output EO at the port permit.EO if permit.PERMIT has the value TRUE. Due to the outgoing connection, this triggers an input EO at counter.CU. A fan-in is demonstrated at the pin permit.EI and a fan-out at qNot.CNF.

3.2.3. Data connections

When a value is issued at an output data port, it needs to be provided to all connected inputs. A fan-in of data connections is not allowed because it would leave undefined which data value is used. Figure 4 visualizes the relationship between the internal and published data. First, the values are only updated internally based on the execution. As soon as an associated output event is issued, this value is published to the connection. Finally, input data is sampled, meaning that the value of the connection is stored as an internal value in the input variable. The sampling process occurs when an EO of an associated event port is processed at the FB. This association is defined as a With model element. The internal value of the data output may thus differ from the one that is available on the connection.

This process is also represented in the running example. The value of counter.Q (i) changes during execution but is (ii) updated only when the EO is issued at counter.CUO, while (iii) qNot.DI1 is updated as soon as an EO qNot.REQ is processed. As a result, the values of counter.Q and qNot.DI1 may be different during the execution.

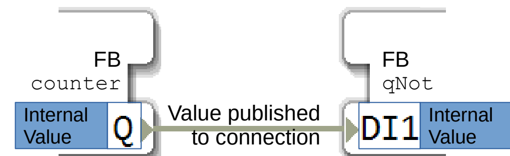


Figure 4 Data values are (i) updated internally during the execution of algorithms, and then (ii) published to the data connection when issuing an associated event occurrence (EO). The receiving FB (iii) updates its internal value when processing an associated EO.

3.2.4. Parameter

A parameter is the specified value that is set at the respective data port and is used for calculations. In the

absence of an input connection at the same port, the parameter acts as a constant. Otherwise, the value serves as an initial value that is overridden as soon as a value is provided via the connection. When a parameter is not provided in the network of FBs, a specified initial value from the type definition has to be considered. If no initial value is specified, the default value of the data type is valid, which is defined in IEC 61131-3.

In the running example, `counter.PV` in Figure 3 is set to the constant 2, while the initial value of the data input at `E_DEFAULT_PERMIT` was set to true in the FB type definition. The value of `counter.Q` is set to true by the FB implementation of `E_CTU` when `counter.CV>=counter.PV` and, thus, depends on the parameter. As `permit` uses this value after it was inverted by `qNot`, `permit` does not issue any further output events and stops the loop.

3.3. Executing IEC 61499 Applications

After the general overview of IEC 61499 execution semantics and relevant model elements, this subsection discusses the execution of IEC 61499 models in different platforms. IEC 61499 applications are usually executed in an RTE that runs on a PLC (Lyu & Brennan 2021). So far, no mechanism for executing the platform-independent Application model is available. The implemented execution semantics for FB networks varies as the standard describes them only textually and in a generic way, leaving room for interpretation. A variety of execution models for FB networks have been described, which are all compliant with IEC 61499 (Dubinin & Vyatkin 2012), but produce different behaviors for an executed model (Ferrarini & Veber 2004). Each RTE realizes a specific execution strategy, which differs mainly in the following aspects (Prenzel et al. 2020): (**i**) Trigger mechanism of the FBs, including the scheduling principle of triggered FBs (FB scan order is either fixed or determined flexibly during the execution) (Ferrarini & Veber 2004), (**ii**) whether FBs in the same execution container can be executed in parallel (multitasking) (Ferrarini & Veber 2004), (**iii**) process of data forwarding and sampling (Pang et al. 2014).

The core goals of IEC 61499 were the interoperability of automation systems, as well as the portability between the tool environments of different vendors (Vyatkin 2011). In practice, portability is limited even for the Application model due to differences in implemented execution semantics, which affects the behavior of an FB network (Pang et al. 2014). A proposed test suite for language features of IEC 61499 FBs has evaluated portability (Xavier et al. 2023) between two actively developed tool environments (Lyu & Brennan 2021). The issues identified concern mainly the porting of the standardized XML format. The test suite comprises FBs that check the support of data types, including boundaries, standard functions, and hierarchical structures (Xavier et al. 2023). Another test suite is available for analyzing the execution semantics of an IEC 61499 RTE with the objective of evaluating simulators (Wiesmayr et al. 2023). It comprises FB networks that check for the semantics that is implemented in an RTE with respect to instances of all kinds of FBs, connections, parameters of FB instances, and hierarchical structures (Composite FBs, subapplications, adapters) (Wiesmayr et al. 2023). As a mechanism to increase portability, dedicated semantics-robust design patterns were proposed by (Dubinin & Vyatkin 2012). However, these design patterns cannot account for all possible variants of the execution semantics (Dubinin & Vyatkin 2012). The platform-specific model, i.e., the Application model mapped to one or more devices, includes hardware access and is never portable (Vyatkin 2011). An abstract model of the execution is defined, for instance, in (Cengic & Akesson 2010), which includes an application execution function e. This function describes how an Application model reacts to an external trigger event and processes it. We do not provide a formal model of the execution semantics, but the developed interpreter could be considered a realization of such a function, as it processes all steps of an Application model until no further events are issued.

In summary, several works classify the execution models implemented in RTEs (Prenzel et al. 2020), or suggest new execution models. Managing variants has not been addressed and approaches that are extensible to multiple variants could facilitate the development process of heterogeneous systems.

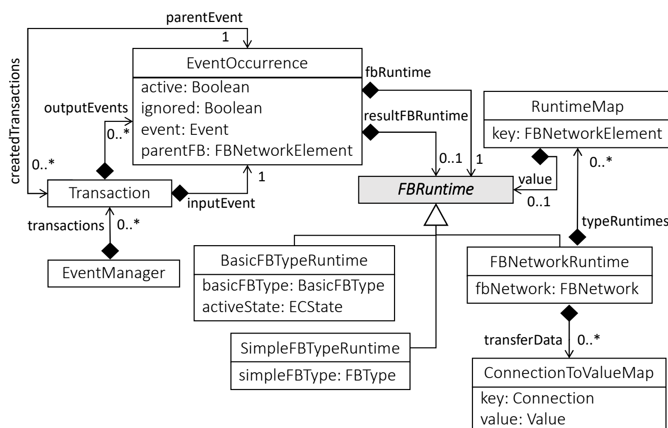


Figure 5 Meta-model of the elements of the interpreter.

4. Interpreter for FB Networks

Formalizing the execution semantics described verbally in the previous section in a meta-model and pseudocode provides a reference implementation of execution environments that are compliant with IEC 61499.

4.1. Interpreter: Meta-Model

We have defined the main domain concepts of the execution semantics in a meta-model (Figure 5), which extends the model elements of IEC 61499 (Figure 2) with semantic information. The `EventManager` holds a list of `Transactions`, which need to be executed. The respective `FBRuntime` is responsible for executing a model element. The execution typically creates new `EventOccurrences` (`outputEvents`). The created Transactions are added to the EventManager for further processing. Interpretation ends when no unprocessed Transactions are remaining. In the following, we describe each concept.

4.1.1. EventOccurrence is an instantaneous trigger that occurs at an `event` port of an FB instance. The instance is

referenced as the `parentFB`. Until the `EventOccurrence` is delivered to an FB instance and processed, the EO remains `active`. An EO triggers only one execution action, e.g., a single transition in the internal state diagram of a Basic FB. If the internal implementation of the FB does not react, the EO is marked as `ignored`.

4.1.2. *Transaction* holds the reaction of an FB instance, which consists of a single `inputEO` triggering this instance and all `outputEOs` that were issued upon this trigger. During the network execution, the outputEOs are transported to any connected FBs, where they act as triggers, hence, initiating a new transaction. Each EO holds a reference to all `createdTransactions` to allow tracing the origin of any scheduled transaction.

4.1.3. *FBRuntime* executes individual IEC 61499 components. As the implementation of an FB instance is defined by its FB type, executing any FB requires interpreting the type declaration. For each kind of FB type, a dedicated runtime is required (e.g., `BasicFBTypeRuntime` from (Wiesmayr et al. 2021)). For this work, also a runtime for executing types with a single encapsulated algorithm (`SimpleFBTypeRuntime`) was created. The newly introduced `FBNetworkRuntime` is responsible for executing a network of FBs according to the rules described in Section 3.2. It maintains the published values of all connections in the `ConnectionToValueMap` and a type runtime per FB instance of the FB network in the `RuntimeMap`. After executing an FB, new transactions are generated for the connected FBs of each output event. Each EO holds references to the state of the model before the execution (`fbRuntime`) and after the execution (`resultFBRuntime`).

4.1.4. *EventManager* is responsible for handling the event queue as a list of transactions. An `EventManager` holds a sequence of transactions over FB instances and is responsible for scheduling them for execution. The order in which the FBs are executed affects the behavior of a network of FBs (Wiesmayr et al. 2023). The event manager offers a method for processing the contained list of `transactions`.

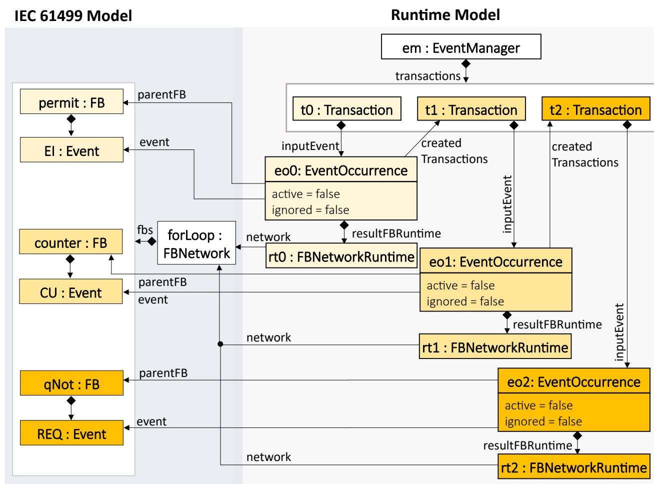


Figure 6 Runtime view of the interpreter for the running example in Figure 3.

4.2. Interpreter: Execution Based on Running Example

The before described execution of an FB network is presented for selected methods in Algorithm 1. This subsection describes the algorithm based on the running example (Figure 3), see Figure 6 for the corresponding runtime view of the interpreter. When interpreting the running example according to the presented concepts, an initial setup is required. The `EventManager` is created, which initially holds one transaction t_0 with the initial trigger event (L 2 in Algorithm 1). For the loop, the `inputEvent` is defined as the active EO, which

```
   Input: eventPort, fbNetwork
1  Function Main:
2      eventManager = createFrom(eventPort,
        fbNetwork) ProcessNetwork(eventManager)
3
4  def ProcessNetwork(em):
5      queue ← Transactions within em
6      forall transaction T_i within queue do
7          createdT ← processTransaction(T_i)
8          Add all createdT to queue
9          if Exists Next Transaction T_{i+1} then
10             result ← T_i.inputEvent.resultFBRuntime
11             T_{i+1}.inputEvent.fbRuntime ← result
12
13 def ProcessTransaction(T):
14     rt ← fbRuntime within T.inputEvent
15     T.inputEvent.resultFBRuntime ← copy(rt)
16     createdT ←
        RunFbNetwork(T.inputEvent.resultFBRuntime,
        T.inputEvent)
17     return createdT
18
19 def RunFbNetwork(rt_NW, eo):
20     rt_FB ← rt_NW.typeRuntimes.get(eo.parentFB)
21     if rt_FB not set then
22         rt_FB ← newly created type runtime
23         typeRuntimes.put(rt_FB)
24     SampleDataInput(rt_FB, rt_NW)
25     typeEOs ← RunType(rt_FB)
26     forall event occurrence eo within typeEOs do
27         write data outputs from rt_FB to rt_NW.network
28     forall event occurrence eo within typeEOs do
29         write data outputs from rt_FB to connected pins
30     forall event occurrence eo within typeEOs do
31         triggeredFbs ← connected FB instances from
            parentFb
32         createdT ← newly created FB transactions
33     return createdT
```

Algorithm 1: Interpreter Execution Algorithm

refers to the `parentFB` permit and the `event` permit.EI. The EO holds a reference to a newly created `FBNetworkRuntime`, which holds a copy of the FB network and empty maps. The following operations are required:

- The event manager determines which transaction to process next (initially t_0) and schedules it for execution (L 7).
- The transaction is processed (L 15-18), which involves executing the FB network (L 17). The network runtime creates a new type runtime for `E_PERMIT`, stores it in the map (L 23-24), and executes the FB for the defined input EO (L 26). All resulting output EOs are collected in t_0 (variable `typeEOs`). Based on the connections, the triggered FBs are identified (L 33) and a new transaction t_1 is created for an EO at the input `counter.CU`. The output EO that caused this transaction stores a reference to t_1 in its list of `createdTransactions` (L 34).
- Next, the event manager schedules t_1 for execution. Its execution also updates the data values. The new FBRuntime for E_CTU is created and stored in the `RuntimeMap` (L 22-24). The algorithm has to update any data inputs based on the incoming connections and parameters, i.e., assign the value 2 to counter.PV (L 25). As `counter.Q` has an outgoing connection, and an associated output EO is issued, its value is stored in the `ConnectionToValueMap` (L 29-30). Transaction t_2 is created for the event `REQ` of instance `qNot` (L 31-33).
- When executing t_2, the data value from the ConnectionToValueMap is provided to the FB type's runtime for any ingoing connection (L 25).
- The execution ends when no more unprocessed transactions are available in the event manager.

Please note that FBRuntimes are only created when they are needed. This significantly reduced the memory usage of the interpreter. Even in large networks, an execution scenario may only affect a small subset of FB instances. FBRuntimes for FB types need to be stored for further execution in the RuntimeMap because the internal state has to be kept.

5. Tool Support

The model execution framework is implemented as part of 4diac IDE from the Eclipse 4diac open source project (Eclipse 4diac 2025). The presented meta-model of the interpreter (Figure 5) is implemented in the Eclipse Modeling Framework (EMF). In addition to the structure and parameters, an operation `process()` was added for each relevant class shown in the meta-model. As part of the model-based execution framework, we provide support for creating, comparing, and visualizing execution traces. It has the following components:

- An interpreter for executing FB networks that follows the execution strategy of 4diac FORTE.
- An API for creating an event manager that includes all required subelements based on a specified trigger event.
- A public API for interacting with execution traces. A processed event manager is seen as a trace that contains a list of transactions. As a copy of the runtime is included in each EO, the history of the execution can be studied.
- Editors for inspecting execution traces in a tree format or in a graphical diagram visualization.
- A trace comparison tool for finding differences between executions.

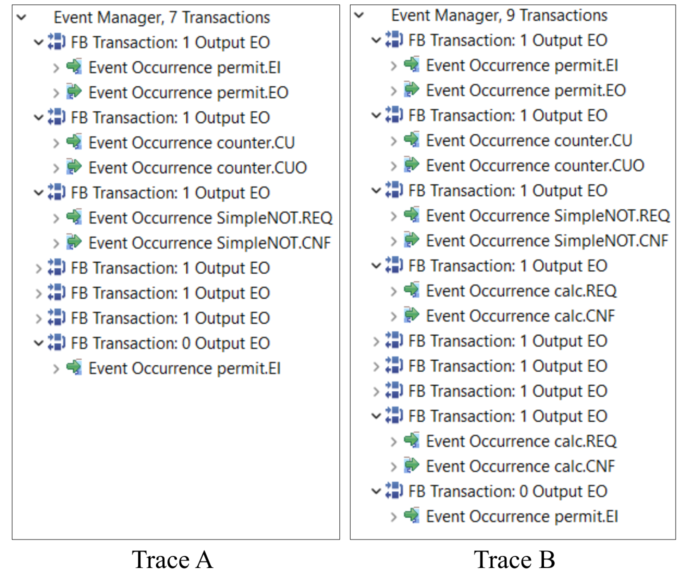


Figure 7 Tree view of execution traces created by interpreting the for-loop (*left*) without content, (*right*) with FB instance `calc` (custom FB) as content. The transactions representing the first loop cycle are shown as expanded in both traces. Trace B has additional transactions for the FB `calc`.

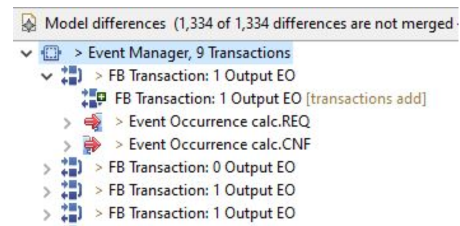


Figure 8 Excerpt of the comparison tool illustrating the difference between Trace A and Trace B of Fig. 7, reflecting the added loop content. Editor is based on EMF Compare.

5.1. Recording Trace from Existing Implementation

A menu entry to record an execution trace is available in 4diac IDE if the user selects the designated pin to trigger. For our running example, a typical trace will start with `permit.EI` which initiates the for-loop. As the structure of the execution traces is defined by the EMF implementation of the meta-model, the available Eclipse infrastructure for saving and loading them in XMI format can be used. We chose the file ending `opsem` for any stored execution traces. Users can access these trace files via a customized tree editor (Figure 7) for analyses. This view reflects the structure of the meta-model, where a transaction has a single inputEO and a list of outputEOs. The tree editor enables engineers to manually inspect the trace in detail, and

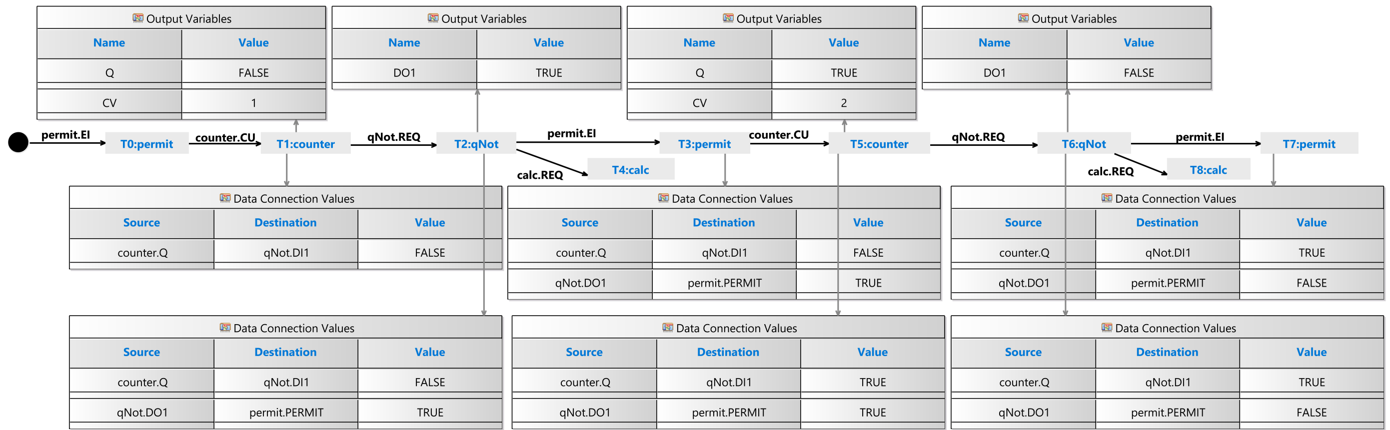


Figure 9 Visualizing a sequence of transactions for the running example of Figure 3 with a connected loop content (FB instance `calc`). For transactions over permit, counter, and qNot (*center*), we show the values of the FB's output data ports (*top*), as well as the current values of data connections (*bottom*). All transactions are executed sequentially, logical branches are explicitly illustrated (e.g., T3 and T4 are both caused by the output EO of `qNot` in T2).

to evaluate whether it matches the expected behavior. It can serve as an extension mechanism for debugging or testing tools, which could provide additional functionality to the editor.

5.2. Comparing Execution Traces

To test an FB network, a developer needs to define a model in XMI format. Alternatively, to enable regression tests (e.g., after refactoring), this model can be recorded from an FB network as described in Section 6.1 and manually inspected for correctness. Successfully executing a test requires compliance of the execution results with the defined model.

In order to automatically provide assertions between generated traces and expected results, we have implemented a custom comparison mechanism in EMF Compare (Brun & Pierantonio 2008). This tool provides out-of-the-box support for matching XMI files, including a graphical comparison editor. By default, EMF Compare performs a graph comparison and does not take the sequential order of the trace elements into account. This is however important in our comparison. For instance, correct access of inputs and outputs may depend on an unaltered order of event triggers. The default matching mechanism of EMF Compare was therefore not sufficient and our custom matcher considers the order of EOs and transactions.

The comparison tool can be accessed in Java and in 4diac IDE. We used our matcher to analyze the effect of changes in the control software and to prepare unit tests for our own interpreter (cf. Section 6.1). An example of comparing the two traces in Figure 7 is shown in Figure 8, where an additional transaction is automatically identified. The visualization enables the identification of differences even between long execution traces, which are difficult to understand via manual inspection. Hence, developers can use it for testing whether the execution of an FB network matches a previously defined trace.

5.3. Visualizing Traces

Representing scenarios as graphical diagrams provides developers with a more intuitive view of the behavior of an FB network. We have implemented this visualization using the graphical framework provided by Eclipse Sirius (Eclipse Sirius 2023). For the running example, the result is shown in Figure 9 with an FB named `calc` attached to `extOut`. Starting from the event occurrence `permit.EI`, each transaction is visualized, illustrating the order of FB execution and its results. Beginning from a start node, the input EO is visualized at the transition. Each transaction is illustrated as a box and lists the name of the parent FB. Additional overlays enable the inspection of the data values. The upper row shows the values of all data ports of the FB. These values are not necessarily published. Therefore, the lower row lists the values published to data connections. Both overlays can be enabled separately for a better overview.

6. Evaluation

We have evaluated our model execution framework with respect to the completeness of the realized set of language features of IEC 61499 and the correctness, i.e., the closeness of simulation results with 4diac FORTE (cf. Subsection 6.1), and the scalability for obtaining longer execution traces (cf. Subsection 6.2). The Application model in Subsection 6.3 provides an application example of using the interpreter during the development.

6.1. Test Suite for FB Networks

For comparing our interpreter to 4diac FORTE, we use the evaluation test suite proposed in (Wiesmayr et al. 2023). The test suite contains examples as standardized IEC 61499 XML files, which are available online.[1] They were designed as reference test cases to identify the implemented execution semantics of RTEs and to evaluate whether a simulator or interpreter closely matches these RTEs. Each example covers one aspect, so that any deviations can be precisely identified. The examples were designed as edge cases so that variations in the execution behavior lead to a different output. Hence, we evaluated our interpreter by comparing the outputs of these examples with

[1] github.com/eclipse-4diac/4diac-examples/tree/master/compliance_tests

those on 4diac FORTE. The test suite currently includes 30 examples and also lists the output provided by 4diac FORTE. The execution traces obtained by the interpreter were manually compared to the expected outputs. They were also inspected for plausibility. The obtained traces were then used as automated JUnit plugin tests to serve as regression tests for further adaptations of the interpreter. The interpreter successfully covered the following aspects:

– Instances of Simple or Basic FBs, and their combination
– Event and data connections with various data types
– Initial values of data types and FB types
– Parameters, including type casts
– Loops, including self-loops across a single FB instance

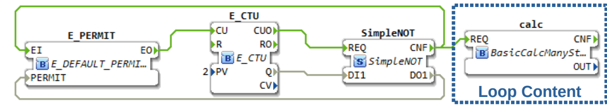
Based on the results of the compliance test suite, we could fix initial implementation errors, e.g., with respect to storing the internal state of FBs. The interpreter does not currently cover FB networks with multiple levels of hierarchy (i.e., subapplications and Composite FB types).

6.2. Scalability Analysis

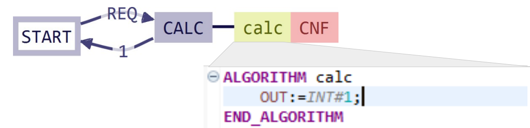
The interpreter complements the execution in an RTE. Considering the effort involved in deploying software to the RTE (i.e., Variant B in Figure 1), we assume that an execution time of the interpreter of up to 2 min is acceptable for early validation even for programs with only a few FBs. We have used variations of the for-loop to analyze the execution time in four different scenarios. Each variant executes a different Basic FB (i.e., an implementation of the custom type `CalcFB`) that represents the loop content. Hence, we can analyze the effect of the complexity of the internal implementation of an FB instance on the execution time of the interpreter. Each Basic FB has one input event port, one output event port, and one data port of type integer. They differ in the implemented state diagram (i.e., the ECC). All variants are triggered by the event occurrence `REQ` and complete their execution in the START state because transitions with condition "1" are always `TRUE` and fire immediately.

– **Simple assignment:** Variant **A** has two states (cf. Figure 10b). Whenever receiving trigger `REQ`, the algorithm `calc` assigns the value 1 to the output port. Event `CNF` is issued.
– **Complex algorithm:** Variant **B** differs in the algorithm `calc`, which calculates the 20th fibonacci number and writes it to the output port (Figure 10c).
– **Many actions:** Variant **C** has ten actions in state `CALC`. Upon receiving event `REQ`, all actions are executed sequentially. The first nine actions perform an addition and assign the result to the output port. The last action issues the output event `CNF` (Figure 10d).
– **Many states:** Variant **D** is functionally equivalent to **C** but the actions are separated into nine different states (Figure 10e). Upon execution of an action, the next state is immediately set active (transition condition is TRUE).

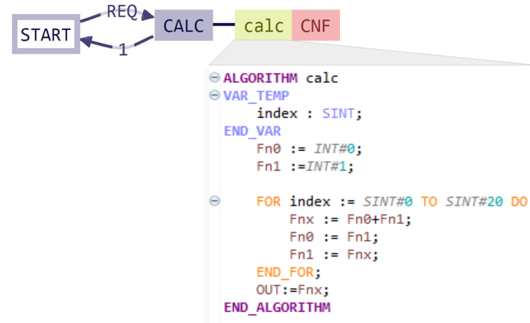
For each variant, the parameter `E_CTU.PV` was set to a value between 1 and 10 000 (i.e., the final loop counter was adapted). Hence, the loop was executed between 1 and 10 000 times,



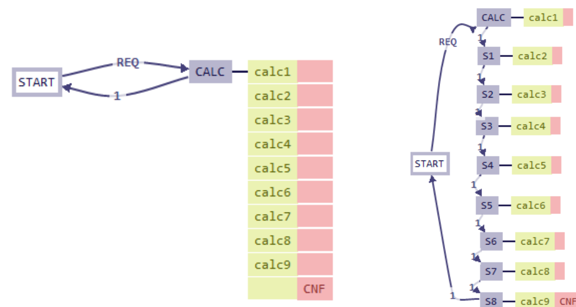
(a) For-loop with attached loop contents (replaced for different variants).



(b) Variant **A**: Simple assignment.



(c) Variant **B**: Complex algorithm.



(d) Variant **C**: Ten actions. **(e)** Variant **D**: Nine states.

Figure 10 Differences between variants **A** to **D**. The implementation complexity affects the execution time.

leading to between 4 and 40 000 executed FB instances (once per loop run `E_PERMIT`, `E_CTU`, `SimpleNOT` and `calc`). The experiment allows assessing the scalability of the interpreter with respect to the length of execution traces, i.e., the number of executed transactions. The experiment was implemented as an Eclipse plugin test that automatically starts the interpretation in 4diac IDE. The execution time was recorded based on a comparison of the system time before and after processing the event manager. The startup phase was not included in the measurement. All measurements were stored in a CSV file. For each configuration, the execution was repeated 25 times (5 executions as warm-up for the Java RTE, 20 executions as evaluation). Materials are available online.[2] The experiments were performed on a PC with an AMD Ryzen 5 3600 6-core processor and 16 GB RAM. Eclipse IDE 2025-09 was used to run the evaluation and was configured to use up to 8100 MB of RAM. It uses Eclipse Temurin JDK 21 for Java.

[2] github.com/bwiesmayr/nwinterpreter

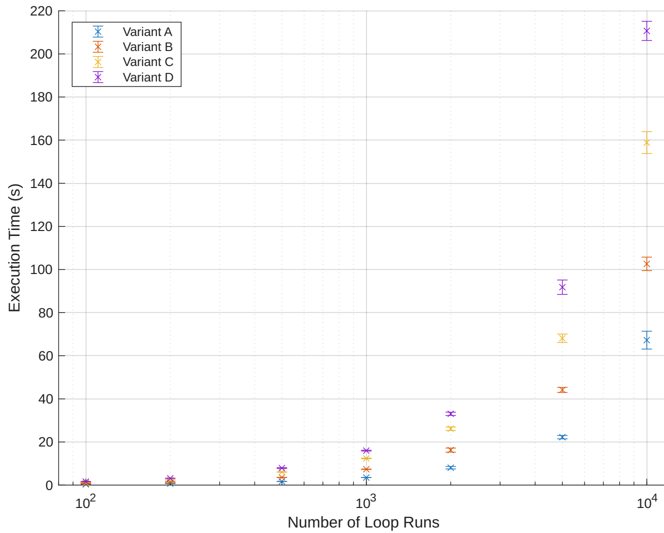


Figure 11 Execution time of variants **A** to **D** for 100-10 000 loop runs.

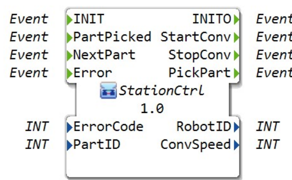


Figure 12 Interface of the `StationCtrl` component.

Table 1 shows the average execution time for all four variants. As expected, the execution time of Variants **C/D** is significantly higher than for Variants **A/B**. Figure 11 additionally illustrates the measurement results and shows that the execution time grows approximately linearly with the number of executed FB instances in all variants. For very long execution traces, the execution time rises more quickly, which we attribute to the high RAM usage. As each interpretation covers the reaction to a single trigger EO, we conclude that interpretation is currently feasible for small- to middle-scale FB networks.

6.3. Application Example

For a more realistic use case, we use the application example from (Wiesmayr et al. 2021), but reimplement the functionality of the state diagram of the core controller FB as a network of FBs. Figure 12 shows the interface of the component with event and data pins, which is equivalent to the one of the FB presented in (Wiesmayr et al. 2021). The main functionality of the component for controlling a station with a conveyor belt and two robots is summarized as follows: An initialization event sets the conveyor belt's target speed to 100 % and a confirmation event `INITO` is issued. When a part is detected, the component receives the event `NextPart`. It selects a robot based on the `PartID` and triggers this robot by issuing the event `PickPart`. Also, the conveyor belt is paused (`StopConv`). If errors occur, the conveyor belt is stopped or its speed is reduced. The implementation shown in Figure 13 mainly uses FBs defined in the IEC 61499 standard, but also includes custom FBs. They are internally realized with state diagrams or Structured Text code.

The interpreter and its accompanying tooling were used to develop and debug the FB network. Traces caused by events that occur at the interface of the `StationCtrl` component were created using our interpreter. For that, we needed to adjust it to handle event occurrences at the interface of a component. We also executed the application on a 4diac FORTE runtime and compared the results to those of the interpreter. The execution was successfully evaluated for equal output data and generated event occurrences. We also measured the execution time for processing the user interaction, executing the interpreter, and saving the trace in a file. For an event trigger at `INIT` (3 transactions), the reaction time was below 0.2 s, for an event trigger at `NextPart` (9 transactions), the reaction time was below 0.35 s. Hence, users receive results almost instantaneously, thus, facilitating the efficient inspection and analysis of FB networks.

6.4. Threats to Validity

This subsection discusses the threats to validity that affect our study based on (Wohlin et al. 2024).

Construct validity. Our analysis and implementation are based on our own examples, rather than using real-world industrial cases. The used examples are, however, based on previous publications (Wiesmayr et al. 2021, 2023) and not specifically designed for the purpose of this study. To assess the functional correctness of executing IEC 61499 models, the examples were designed to cover edge cases and a variety of language features. However, the application example (StationCtrl component, Section 6.3) is specifically adapted for this study, i.e., the functionality previously realized in a state diagram was implemented as a network of FBs. For the performance evaluation, we chose a loop with four different calculation variants used as loop content. By adjusting the number of loop runs, we could directly assess the performance of the interpreter in relation to the number of executed FB instances. The used loop example constitutes a small-scale model, however, it contains a typical behavior of automation software, namely to execute a piece of code repeatedly (e.g., cyclically every n ms). The loop example also enabled us to evaluate multiple variants of one FB to assess the influence of certain characteristics of the executed FBs.

Conclusion validity. The extent of the performance measurement does not allow for detailed statistical analyses. However, the measurement results indicate a linear relation between execution time and number of executed instances.

Another threat to conclusion validity is the manual comparison of the execution traces with the results observed in the runtime 4diac FORTE. As the interpreter focuses on the causality of event occurrences rather than predicting timing behavior, the equivalence of execution traces requires manual interpretation. We believe that the order of executed transactions is more important than the precise timing for most practical cases which require early validation of models. To mitigate this threat, the manually crafted examples from (Wiesmayr et al. 2023) served as the main comparison target. They were designed to show different results depending on the execution behavior that is realized in an RTE or an interpreter. This allowed us to focus on

Table 1 Measurement results for the execution duration t_{avg} of the for-loop with varying complexity of the loop content FB.

Variant	t_{avg} for PV:=1	t_{avg} for PV:=1000	t_{avg} for PV:=10 000
A Single Assignment	(16.26±1.27) ms	(3.49±0.03) s	(67.23±4.14) s
B Fibonacci number	(17.83±0.69) ms	(7.36±0.10) s	(102.61±3.12) s
C Ten actions	(64.97±18.40) ms	(12.38±0.15) s	(158.86±5.07) s
D Nine states	(102.82±60.22) ms	(15.99±0.19) s	(210.67±4.44) s

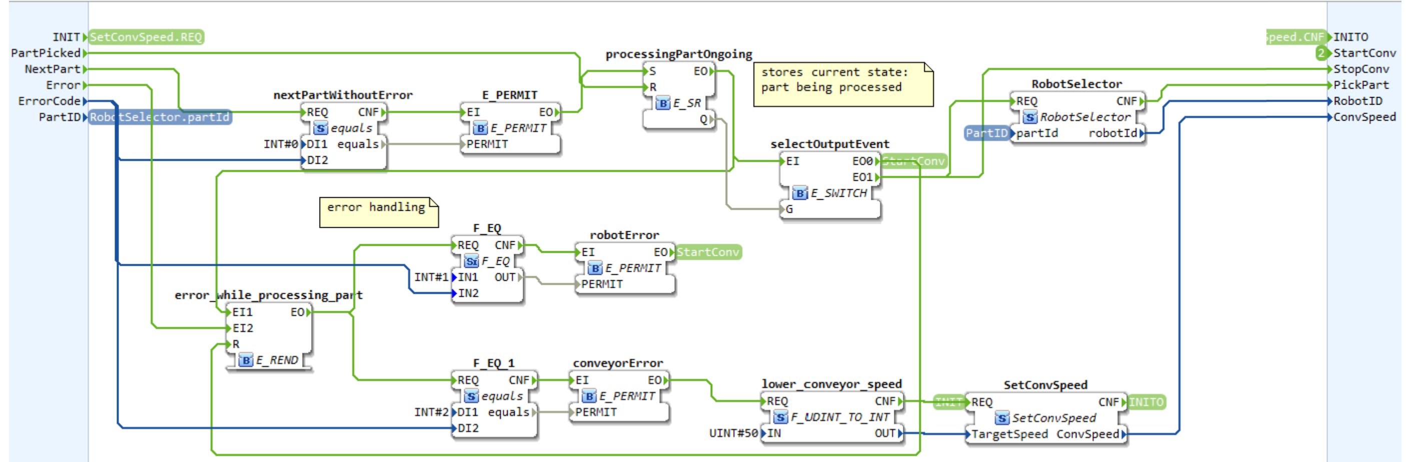


Figure 13 `StationCtrl` Network with 12 Function Blocks. Some event and data lines are hidden to improve readability; their endpoints are shown as labels. The interface bars (shadowed in blue) show the interface pins of the `StationCtrl` component.

the results of the calculations. In addition, multiple variants of one FB were created, as this allowed us to assess the influence of certain characteristics of the executed FBs on the performance of the interpretation. However, the used loop example does not reflect all aspects of typical control functionality.

Internal validity. The internal validity of the performance measurement is affected by the limited precision of capturing the current time. The interpreter is part of an Eclipse Rich Client Platform and not a stand-alone tool. Hence, Java benchmarking tools could not be used and the benchmark had to be manually implemented as an Eclipse Plugin Test. This may affect the validity of the results. As the interpreter is implemented in Java, performance measurements may be affected by the warm-up phase of the Java RTE. To mitigate this risk, we discarded the first five measurements for each configuration.

External validity. We used a small set of hand-crafted examples and the presented results cannot be generalized for other models, e.g., large-scale models or systems requiring a huge number of events. A case study with industrial examples would be necessary to assess the usefulness of the interpreter in real-world automation projects.

While the workflow is sped up compared to deploying software to an RTE and executing it within the RTE, and the measurement results indicate a high performance of the interpreter, we cannot be certain that the performance is sufficient for real industrial cases. A user study could identify typical usage of the interpreter by developers and thus the required performance.

7. Conclusions & Future Work

Interpreting control software offers possibilities for early feedback during development. Furthermore, it allows to execute the platform-independent Application model rather than deployed code. Interpretation is a well-known technique in the domain of model-driven engineering, but had not been applied to control software based on FB networks that are compliant with IEC 61499. Executing FB networks is the focus of this paper. Rather than running the platform-specific model in an RTE, the platform-independent FB networks are analyzed.

Based on an analysis of the IEC 61499 standard, we identified the relevant language elements for the interpretation of FB networks. Our implementation in EMF and Java allows to interpret FB networks that consist of Basic FBs, Simple FBs, connections, and parameters. The execution semantics vary in the order of the executed FBs, the handling of parallel events, and the sampling of data values. Our model execution framework focuses on replicating the behavior of 4diac FORTE. Furthermore, we outlined the applicability of the model execution framework for inspecting FB networks and for testing. The trace comparison tool allows to identify differences in the execution traces. Manual inspection of these differences allows for evaluation of the effect of changes in the software due to refactoring or added/deleted features.

In future work, the model execution framework will need to be extended to cover FB networks with multiple hierarchy

levels to enable the interpretation of large-scale FB networks with structuring mechanisms. As the execution time increased approximately linearly with the number of FB instances, future performance improvements of individual components of the interpreter may increase the applicability for medium-scale control software. Debugging tools could utilize our interpreter to allow step-by-step inspection of execution traces. Developers might also benefit from additional visualization options, which should be assessed in usability tests. If a higher performance is required, traces could be saved iteratively or model splitting techniques can be used. Interactive debugging tools will be required to localize faults. Such tools have been generically proposed for DSLs such as (Enet et al. 2024) and can possibly be adapted for IEC 61499. The current implementation realizes an interpreter based on the variant implemented in 4diac FORTE, which is open source. As variability aspects were considered during the implementation, an adaptation for further variants of the execution semantics is possible.

Acknowledgments

B. Wiesmayr has received funding (LIT project SOFIA, LIT-2024-13-YOU-117) from the Johannes Kepler University Linz, Linz Institute of Technology (LIT), the State of Upper Austria and the Federal Ministry of Education, Science and Research. This research was partly funded by the Austrian Science Fund (FWF) [10.55776/PIN3686425].

References

Akifev, D., Liakh, T., Ovsiannikova, P., Sorokin, R., & Vyatkin, V. (2023). Debugging approach for IEC 61499 control applications in fbme. In *IEEE 32nd Int. Symp. on Industrial Electronics (ISIE).* doi: 10.1109/ISIE51358.2023.10228129

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice* (2nd ed.). Morgan & Claypool.

Brun, C., & Pierantonio, A. (2008). Model differences in the Eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Buzhinsky, I., Ulyantsev, V., Veijalainen, J., & Vyatkin, V. (2015). Evolutionary approach to coverage testing of IEC 61499 function block applications. In *13th Int. Conf. on Ind. Inf.* (pp. 1213–1218). doi: 10.1109/INDIN.2015.7281908

Cabral, J., & Zoitl, A. (2025). A systematic literature review of the characteristics of automation systems that impact debugging of PLC software. *IEEE Trans. Automation Science and Engineering*, *22*, 14306-14316. doi: 10.1109/TASE.2025.3559790

Cengic, G., & Akesson, K. (2010). On formal analysis of IEC 61499 applications, part A: Modeling. *IEEE Trans. Ind. Inf.*, *6*(2), 136-144. doi: 10.1109/TII.2010.2040392

Colaço, J.-L., Mendler, M., Pauget, B., & Pouzet, M. (2023). A constructive state-based semantics and interpreter for a synchronous data-flow language with state machines. *ACM Trans. Embed. Comput. Syst.*, *22*(5s). doi: 10.1145/3609131

Dubinin, V. N., & Vyatkin, V. (2012). Semantics-robust design patterns for IEC 61499. *IEEE Trans. Ind. Inf.*, *8*(2), 279-290. doi: 10.1109/TII.2012.2186820

Eclipse 4diac. (2025). *Eclipse 4diac 3.0 - The Open Source Environment for Distributed Industrial Automation and Control Systems.* eclipse.dev/4diac.

Eclipse Sirius. (2023). *Eclipse Sirius.* https://eclipse.dev/sirius.

Elekes, M., Molnár, V., & Micskei, Z. (2023). Assessing the specification of modelling language semantics: A study on UML PSSM. *Software Quality Journal*. doi: 10.1007/s11219-023-09617-5

Enet, J., Bousse, E., Tisi, M., & Sunyé, G. (2024). dpDebugger: a domain-parametric debugger for DSLs using DAP and Language Protocols. In *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems* (p. 71–75). ACM. doi: 10.1145/3652620.3687794

Exelmans, J., Van Mierlo, S., & Vangheluwe, H. (2022). A statecharts interpreter and compiler with semantic variability. In *25th Int. Conf. Model Driven Engineering Languages and Systems: Companion Proc.* (pp. 722–727). ACM.

Ferrarini, L., & Veber, C. (2004). Implementation approaches for the execution model of IEC 61499 applications. In *2nd IEEE Int. Conf. on Ind. Inf.* (p. 612-617). doi: 10.1109/INDIN.2004.1417418

Guellouz, S., Benzina, A., Khalgui, M., Frey, G., Li, Z., & Vyatkin, V. (2019). Designing efficient reconfigurable control systems using IEC61499 and symbolic model checking. *IEEE Trans. Automation Science and Engineering*, *16*(3), 1110-1124. doi: 10.1109/TASE.2018.2868897

Hametner, R., Hegny, I., & Zoitl, A. (2014). A unit-test framework for event-driven control components modeled in IEC 61499. In *IEEE Int. Conf. on Emerging Technology and Factory Automation.* doi: 10.1109/ETFA.2014.7005209

Hametner, R., Kormann, B., Vogel-Heuser, B., Winkler, D., & Zoitl, A. (2013). Automated test case generation for industrial control applications. In *Recent advances in robotics and automation* (pp. 263–273). Springer. doi: 10.1007/978-3-642-37387-9_20

Huang, Y., Bu, X., Zhu, G., Ye, X., Zhu, X., & Shi, J. (2019). KST: Executable formal semantics of IEC 61131-3 Structured Text for verification. *IEEE Access*, *7*, 14593-14602. doi: 10.1109/ACCESS.2019.2894026

IEC. (2025). *IEC 61131 - Programmable controllers, Part 3: Programming languages.* International Electrotechnical Commission.

IEC TC65/WG6. (2012). *IEC 61499-1, Function Blocks, Part 1: Architecture.* International Electrotechnical Commission.

Liakh, T., Sorokin, R., Akifev, D., Patil, S., & Vyatkin, V. (2022). Formal model of IEC 61499 execution trace in FBME IDE. In *20th Int. Conf. on Ind. Inf.* (p. 588-593). doi: 10.1109/INDIN51773.2022.9976176

Lyu, G., & Brennan, R. W. (2021). Towards IEC 61499-based distributed intelligent automation: A literature review. *IEEE Trans. Ind. Inf.*, *17*(4), 2295–2306. doi: 10.1109/TII.2020.3016990

Object Management Group. (2015). *Unified Modelling Language: 2.5.* https://www.omg.org/spec/UML/About-UML/.

Object Management Group. (2017). *Action Language for Foundational UML.* https://www.omg.org/spec/ALF/.

Object Management Group. (2021). *Semantics of*

a Foundational Subset for Executable UML Models. https://www.omg.org/spec/FUML/.
Pang, C., Patil, S., Yang, C. W., Vyatkin, V., & Shalyto, A. (2014). A portability study of IEC 61499: Semantics and tools. In *12th IEEE Int. Conf. Industrial Informatics* (pp. 440–445). doi: 10.1109/INDIN.2014.6945553
Prenzel, L., Zoitl, A., & Provost, J. (2020). IEC 61499 runtime environments: A state of the art comparison. In *Computer Aided Systems Theory – EUROCAST* (pp. 453–460).
Prähofer, H., Schatz, R., Wirth, C., & Mössenböck, H. (2010). Deterministic replay debugging of IEC 61131-3 SoftPLC programs. In *8th IEEE Int. Conf. on Ind. Inf.* (p. 1110-1117). doi: 10.1109/INDIN.2010.5549586
Schütz, D., Legat, C., & Vogel-Heuser, B. (2014). MDE of manufacturing automation software — integrating SysML and standard development tools. In *IEEE Int. Conf. on Ind. Inf.* (p. 267-273). doi: 10.1109/INDIN.2014.6945519
Sehr, M. A., Lohstroh, M., Weber, M., Ugalde, I., Witte, M., Neidig, J., ... Lee, E. A. (2021). Programmable logic controllers in the context of industry 4.0. *IEEE Trans. Ind. Inf.*, *17*(5), 3523-3533. doi: 10.1109/TII.2020.3007764
Sinha, R., Patil, S., Gomes, L., & Vyatkin, V. (2019). A survey of static formal methods for building dependable industrial automation systems. *IEEE Trans. on Ind. Inf.*, *15*(7), 3772-3783. doi: 10.1109/TII.2019.2908665
Suzuki, G., Watanabe, T., & Moriguchi, S. (2023). Implementation and evaluation of an interpreter for functional reactive programming on small embedded devices. In *Companion Proc. of the 7th Int. Conf. on the Art, Science, and Engineering of Programming* (p. 12–16). ACM. doi: 10.1145/3594671.3594674
Ulewicz, S., Vogel-Heuser, B., Simon, H., Bohlender, D., Obster, M., & Kowalewski, S. (2017). A priori test coverage estimation for automated production systems: Using generated behavior models for coverage calculation. In *22nd IEEE Int. Conf. on Emerging Technologies and Factory Automation.* doi: 10.1109/ETFA.2017.8247704
Vyatkin, V. (2011). IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *IEEE Trans. Ind. Inf.*, *7*(4), 768-781. doi: 10.1109/TII.2011.2166785
Vyatkin, V. (2013). Software Engineering in Industrial Automation: State-of-the-Art Review. *IEEE Trans. Ind. Inf.*, *9*(3), 1234–1249. doi: 10.1109/TII.2013.2258165
Wachsmuth, G. (2007). Modelling the operational semantics of domain-specific modelling languages. In *Int. Summer School Generative and Transformational Techniques in Software Engineering* (pp. 506–520). Springer.
Wenger, M., Melik-Merkumians, M., Hegny, I., Hametner, R., & Zoitl, A. (2011). Utilizing IEC 61499 in an MDA control application development approach. In *Int. Conf. Automation Science and Engineering (CASE)* (p. 495-500). IEEE. doi: 10.1109/CASE.2011.6042458
Werner, B., Vogel-Heuser, B., Ziegltrum, S., Grobl, H., & Botzenhardt, C. (2020). Supporting troubleshooting in machine and plant manufacturing by backstepping of PLC-control software. In *IEEE Conf. on Ind. Cyberphysical Systems* (pp. 242–249). doi: 10.1109/ICPS48405.2020.9274778
Wiesmayr, B., Mehlhop, S., & Zoitl, A. (2023). Close enough? criteria for sufficient simulations of IEC 61499 models. In *19th IEEE Int. Conf. Automation Science and Engineering.*
Wiesmayr, B., Zoitl, A., Garmendia, A., & Wimmer, M. (2021). A Model-based Execution Framework for Interpreting Control Software. In *26th Int. Conf. Emerging Technologies and Factory Automation.* IEEE.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2024). *Experimentation in software engineering* (Vol. 2). Springer.
Xavier, M., Laikh, T., Patil, S., & Vyatkin, V. (2023). Developing a test suite for evaluating IEC 61499 application portability. In *2023 IEEE 32nd Int. Symp. Industrial Electronics.* doi: 10.1109/ISIE51358.2023.10228154
Zhou, N., Li, D., Vyatkin, V., Dubinin, V., & Liu, C. (2022). Toward dependable model-driven design of low-level industrial automation control systems. *IEEE Trans. Automation Science and Engineering*, *19*(1), 425-440. doi: 10.1109/TASE.2020.3038034
Zoitl, A., & Lewis, R. W. (2014). *Modelling control systems using IEC 61499* (2nd ed., Vol. 95). London: IET.
Zoitl, A., & Vyatkin, V. (2009). IEC 61499 architecture for distributed automation: The glass half full view. *IEEE Industrial Electronics Magazine*, *3*(4), 7–23. doi: 10.1109/MIE.2009.934789

About the authors

Bianca Wiesmayr is a postdoctoral researcher at the Institute of Business Informatics - Software Engineering and a junior professor for software engineering at the Ulm University. Her main research topic is model-driven software engineering for industrial automation. You can contact the author at bianca.wiesmayr@uni-ulm.de.

Antonio Garmendia is Assistant Professor at the King Juan Carlos University, Spain. Before, he was a postdoctoral researcher at the WIN-SE department, JKU Linz. His research interests are in scalability in model-driven engineering and the construction of graphical modelling environments.You can contact the author at antonio.garmendia@urjc.es.

Alois Zoitl is full professor at the LIT Cyber-Physical Systems Lab of the Johannes Kepler University Linz, Austria. His research interests are in the area adaptive production systems, distributed control architectures, and dynamic reconfiguration of control applications as well as software development and software quality assurance methods for industrial automation. You can contact the author at alois.zoitl@jku.at.

Manuel Wimmer is full professor and head of the Institute of Business Informatics - Software Engineering at the Johannes Kepler University Linz. His research interests comprise the foundations of software engineering techniques and their application in domains such as software modernization, cyber-physical systems, and quantum computing. You can contact the author at manuel.wimmer@jku.at.