A Model-based Execution Framework for Interpreting Control Software

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

Antonio Garmendia, Manuel Wimmer
CDL-MINT
Johannes Kepler University Linz
Linz, Austria
{antonio.garmendia, manuel.wimmer}@jku.at

***Abstract*—Industrial standards define domain-specific languages that are frequently used for developing control software. For instance, IEC 61499 standardizes a graphical modeling language that includes a platform-independent application model. The application is composed of Function Blocks. A runtime can execute the model by implementing the semantics that is described in the standard in natural language. By defining an interpreter for IEC 61499 models, we can directly execute them without prior code generation. This enables providing feedback directly on the model level. We present an interpreter for Basic Function Blocks, which encapsulate a state-based Execution Control Chart. An existing EMF meta-model for IEC 61499 was extended with an operational semantics implemented in Java and Xtend. The test cases are defined either in Java or as an interface model. Such a model is standardized in IEC 61499 as Service Sequences. We evaluate our interpreter by executing the Basic Function Blocks that are defined in the standard and compare our results to those of the open-source runtime 4diac FORTE. As a practical use case, we show how developers can use the interpreter for unit testing self-defined Basic Function Blocks.**

***Index Terms*—Industrial automation, IEC 61499, Operational semantics, Interpreter, Model-driven engineering**

I. Introduction

Smart factories bridge the gap between automation and customization. They comprise a rising number of components that are highly autonomous, distributed and operate in parallel, resulting in so-called Cyber-Physical Systems of Systems. The effort for automation software within a system is therefore rising [1] while the software is also getting more complex [2], [3]. Hence, the use of Model-Driven Engineering (MDE) methodology was reported as a promising practice for developing such systems [4]–[6].

The industrial standard IEC 61499 [7] was developed to address many modern challenges in engineering control applications for distributed systems. The language defines a graphical domain-specific modeling language (DSML) that ensures a component-based design. Functionality is encapsulated within individual software components, so-called Function Blocks (FBs) [8]. The event-based execution semantics is described in natural language and allows building a runtime environment that can run on any device. This runtime executes the IEC 61499 system model and, thus, abstracts the hardware from the software. A platform-independent IEC 61499 Application can be deployed to a single runtime or distributed across several runtimes. IEC 61499 models can be simulated in a runtime directly on the PC, and thus, early feedback and evaluation is feasible before deploying the software to automation devices. As IEC 61499 is designed for interoperability and portability, a distributed application may be executed on runtimes of several vendors. A runtime is also required for testing FBs or parts of applications [9]. Evaluating the platform-independent IEC 61499 Application directly is currently not supported, but may facilitate the development and enhance portability. Integrated Development Environments (IDEs) could provide early feedback to the developer based on the model, and the Application could be implemented and tested without considering potential restrictions of the involved runtime environments. Reuse of FBs across platforms could be thus improved. In early phases of the development, the platforms used in the system may be furthermore subject to change. Finally, a semi-formal definition of the execution semantics of IEC 61499 may avoid differing interpretations among vendors.

In this paper, we present a model-based framework for interpreting control software. We analyzed in detail the execution semantics of IEC 61499 Basic FBs and explain it based on a running example. We implemented our framework in Java and Xtend [10] based on the Eclipse Modeling Framework (EMF) atop the Eclipse IDE. Pseudo-code illustrates the functionality of the event management, as well as the interpreter. We review approaches for unit tests of Basic FBs and apply our interpreter to model-driven testing based on Service Sequences. Finally, we evaluate our approach based on our running example and the Basic FBs that are defined in the standard, and conclude with an outlook on future work.

II. Background and Related Work

A. Related Work

Several research works have successfully applied the MDE paradigm in the development of automation systems. For example, Alvarez et al. [11] propose a methodological approach to employ MDE in order to improve the development process of control systems. In this sense, the authors describe the benefit of using MDE in the growing complexity of industrial automation. Other works have successfully applied MDE techniques to software projects for industrial automation. Vogel-Heuser et al. [12] propose an extension of the Systems

Modeling Language (SysML [13]) to support automation tasks. The authors called their extension SysML-AT (SysML for automation) and, based on this language, they implemented a code generator for runtime environments conforming to IEC 61131-3. Our approach focuses on IEC 61499, given its platform-independent approach compared to the IEC 61131-3 standard. Control programs based on IEC 61131-3 only run on dedicated hardware and are therefore often tested directly at the production plant [9].

The Object Management Group (OMG) propose the Foundational UML (fUML)[1] language for model execution. This language is a subset of the Unified Modeling Language (UML) and covers elements with a concrete definition of its semantics. Additionally, the OMG standardized the Action Language for fUML (Alf)[2], which is a high-level language for specifying executable behaviour. This approach is similar to ours. We propose a language for the operational semantics of the standard IEC 61499 and the implementation for its execution.

B. Executing IEC 61499 models

A Function Block (FB) Type is a definition of a class-like modular software component. Function Blocks are instantiated in a graphical application model and interconnected via signal paths. IEC 61499 supports several kinds of FBs. For each FB type, an interface description and an implementation of the internal behavior are required. While the former is common to all kinds of FBs, the latter differs widely to support multiple use cases: simple algorithms, state diagrams, or whole application parts are feasible. State diagrams are found in Basic FBs (BFBs). In the following sections, we limit our considerations to BFBs according to Edition 2 of the standard. This revised edition improves deterministic execution of BFBs across platforms. IEC 61499 standardizes both a graphical and a textual concrete syntax. Furthermore, IEC 61499 part 2 defines an XML-format for model exchange.

Domain-specific models can be executed either directly (interpretation), or after generating and compiling intermediate code (compilation) [14]. New functionality for IEC 61499 is implemented either by developing new FB types, or by connecting instances of existing FB types into an FB network. Both ways involve *code generation*: FB Types are exported as code in a general-purpose language and integrated into a runtime during the compile process. Then, FB Networks can be deployed to this runtime based on a generated list of commands. The standard defines a management model with the concrete syntax for nine management commands, including `create` and `delete` [8]. When executing these commands, FBs are dynamically instantiated in the runtime and connections are created. Hence, FB networks are *interpreted* by the runtime. Management FBs can also generate events when an execution container is started or stopped [8].

Possible development processes for FB Types are shown in Fig. 1. Advanced tool support is required to seamlessly integrate these processes into an IDE. A typical development process (1) involves generating code to a general-purpose language, compiling the runtime and launching it. The FB Type can then be tested in this runtime instance. The process can be simplified (2) by generating code for an interpretable language. This requires an interpreter within the runtime, but the compilation is omitted, thus reducing the round-trip engineering time for FB Types. When (3) interpreting the model directly, no code generation is required. Platform-independent development of FB Types is thus feasible, which can enhance the possible engineering and testing methodology.

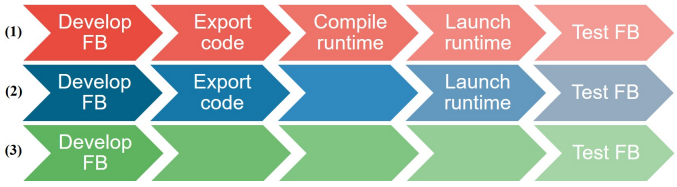

Fig. 1: Development process for FB Types with (1) Code Export and Compile Process, (2) Export to interpretable code, and (3) a Model Interpreter.

Both runtimes and IDEs for IEC 61499 are available from several vendors. Eclipse 4diac [15] is an open-source solution that provides the runtime 4diac FORTE, which is developed in C++ and runs on various platforms (e.g., Linux, Windows, Raspberry Pi, or commercial Programmable Logic Controllers). For FB types, C++ code is generated from the IEC 61499 models and the FBs are compiled as a part of the 4diac FORTE source code (process 1). Additionally, FB types can be exported as interpretable LUA code (process 2). This allows loading types dynamically to an existing instance of 4diac FORTE. As an IEC 61499 runtime can be executed on any PC or any automation device [7], the software remains hardware-independent. The FB and application development is however specific to the runtime. Interpreting Applications or testing FBs without a runtime (process 3) is currently not supported, hence, test cases for FBs need to be implemented in an IEC 61499 Application. The interface of a FB under Test has to be instrumented by another FB that implements a test sequence. Additionally, the Application needs to record the output events and output data of the FB under Test [16]. Model-based approaches can reduce the effort of implementing test cases. The test application is then generated from a dedicated specification, such as UML Statecharts [17] or IEC 61499 models [9].

III. IEC 61499 Basic Function Block By-Example

A large part of typical IEC 61499 applications is composed of Basic Function Blocks (BFBs), which are well-suited for orchestrating the execution of other software parts. Their internals are comprised of a state diagram (Execution Control Chart, ECC) and any number of algorithms. The standard does not define an implementation language for internal algorithms, but typically the programming languages defined in IEC 61131 part 3 are used.

[1] https://www.omg.org/spec/FUML/
[2] https://www.omg.org/spec/ALF/

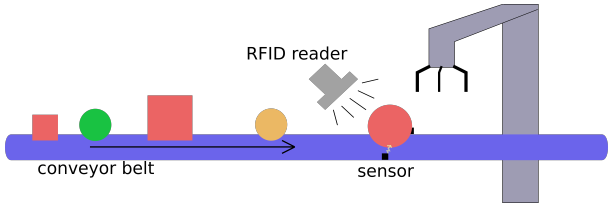

Fig. 2: Abstracted mechatronic station where a robotic arm picks parts from a conveyor belt. The machine recognizes the various kinds of parts based on their RFID tag.

A. Running example

The execution of IEC 61499 BFBs will be explained based on a running example. We introduce the simple mechatronic station illustrated in Fig. 2. In our station, parts are processed by several robotic arms. Different kinds of parts arrive on a conveyor belt and are detected by a motion sensor. Each part is identifiable via its part ID that is stored on an RFID tag. This information is captured by the RFID reader and a suitable robotic arm is assigned to pick the part from the conveyor belt. A simplified BFB orchestrating the example station will be described in this section to illustrate relevant language concepts.

B. Interface Definition

Fig. 3 shows the interface of our example BFB that follows the graphical syntax of IEC 61499. As FBs are fully encapsulated, an instance may only exchange information with its environment via defined interface pins. IEC 61499 supports two kinds of signals: data signals hold data of any type (e.g., Boolean, int, real, time), while event signals occur as discrete event objects. The type name (StationCtrl) is displayed in the center of the block. We designed our BFB with four event inputs (left) and outputs (right). For each pin, we define (i) a valid identifier as name, (ii) the data or event type, and (iii) an optional comment. It is worth noting that changes of data signals are not immediately accessible within an FB. The values are only recorded as soon as an associated event occurs. Changes to output data are only made available to connected FBs upon sending an associated output event. Data pins without an association effectively represent constant values, but unassociated event pins are frequently used. The

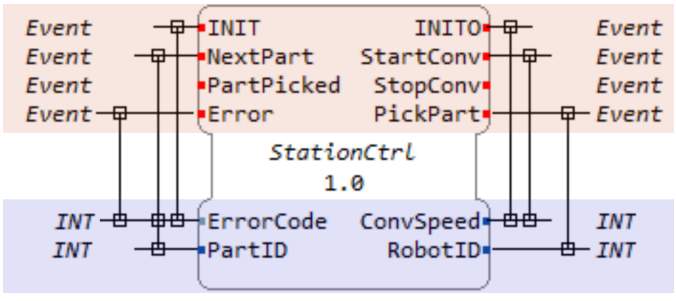

Fig. 3: Interface Specification of the BFB StationCtrl consisting of event pins and data pins. Vertical lines represent the association between event and data (WITH-construct).

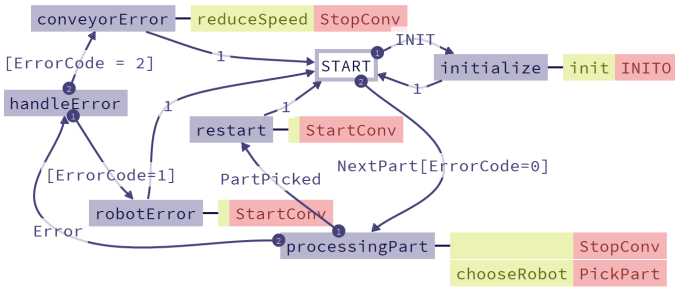

Fig. 4: Execution Control Chart (ECC) for the BFB StationCtrl. States (violet) have entry actions with algorithms (green) and output events (red). The algorithms set the values of the output data: init sets a default conveyor speed, reduce speed halves the current speed, and chooseRobot selects a robot based on the PartID.

association is modeled as a so-called WITH-construct and is graphically represented as a vertical connection (cf. Fig. 3).

C. Internal Behavior

After designing the interface, the expected behavior of the BFB is defined. The Execution Control Chart (Fig. 4) manages sending output events and computing algorithms based on the internal state of the BFB. The initial state of a BFB is the START state, which is marked by a double border. For each state, multiple outgoing transitions are permitted. Transitions are evaluated in order of their priority, starting from priority 1. This ensures deterministic execution, even if multiple transition conditions evaluate to true. Several types of transition conditions are supported:

- 1-Condition: This condition is always true and is crossed immediately upon evaluation (i.e., `1`). It must not be used as an outgoing transition of the START state.
- Event Condition: The transition is crossed upon occurrence of the specified input event (e.g., `INIT`).
- Guard Condition: The transition is crossed when the condition evaluates to true (e.g., `[ErrorCode = 1]`).
- Combined Condition: Upon occurrence of the specified event, the guard condition is evaluated. The transition is crossed if the guard condition holds (e.g., `NextPart[ErrorCode = 0]`).

States can have actions, which are always triggered on state entry. An action is comprised of computing an algorithm and/or sending an output event. Output events can therefore only occur as a response to an input event. An FB is inactive until an input event triggers the FB execution. Hence, timed transitions cannot be specified (such as “after 5 ms”). After receiving a trigger, the FB must follow a “run-to-completion”-semantics. The FB execution can not be interrupted by additional events. The execution container of an application must ensure that no more than one event at a time can be passed to a BFB. After an event trigger, the transitions outgoing from the currently active state are evaluated in order of their priority. Possible cases are:

- No condition holds. The event is nevertheless consumed immediately. Hence, it is ignored.
- A condition holds and the transition is crossed. The input event was consumed. Beginning from this new state,

all outgoing transitions are immediately evaluated again. Only 1-transitions or transitions with Guard Condition can evaluate to true, because the trigger event was already consumed. The evaluation continues until a stable state is reached, where all outgoing transitions evaluate to false.

As an example, assume that the state `START` of our StationCtrl-FB is active. Let an event arrive at the `INIT` input. The transition with priority 1 evaluates to true and the event is consumed. The State `initialize` is active. Algorithm `init` is computed and an event is sent at the output `INITO`. Now the outgoing transitions of `initialize` are evaluated. The transition condition `1` always evaluates to `true`, therefore, the transition is immediately crossed. State START is active again. As the event `INIT` is already consumed, no condition of an outgoing transition holds. The BFB now remains inactive until the next input event arrives. A new input event at the pin `Error` does not allow crossing a transition. The event is ignored. Now, assume an event arriving at the pin `NextPart`. If the guard condition holds, the transition with priority 2 is crossed. Upon entering state `processingPart`, two actions are executed. For each action, the algorithm is first computed before sending the event. Multiple actions are executed from top to bottom. The outgoing transitions are evaluated, but no transition condition holds. Note that the `Error` event has been dismissed and cannot trigger the transition of priority 2.

Unlike other state diagrams, such as UML statecharts [18], the ECC currently does not support parallelism. Only one state can be active at a time. Furthermore, no hierarchical elements are allowed within the ECC. Hierarchy can only be realized by instantiating FBs in so-called Composite FBs or Subapplications. Extensions for hierarchical and parallel ECCs have been proposed [19], but are not part of the IEC 61499 standard.

IV. Approach

Our framework is designed for the model-based execution of IEC 61499 FBs. The interpreter for BFBs has to implement the semantics of the ECC and also requires an interpreter for the algorithms. We focus on algorithms that are defined using Structured Text (ST), which is a textual language standardized in IEC 61131-3. Our execution framework comprises the following components to establish an efficient and effective development and testing process:

- an interpreter for BFBs with ST algorithms,
- infrastructure for managing the events for execution, and for recording the results of each executed event,
- infrastructure for setting up test cases,
- a converter for creating test specifications from IEC 61499 Service models, and
- automated evaluation whether the output events and data conform to the expected outputs of a BFB.

An interpreter traverses the model and executes actions that are defined for the respective object [20]. For instance, a transition of the ECC fires if its condition holds. The corresponding semantics involves evaluating the transition condition and setting the destination of the transition as the new active state of the ECC.

A. IEC 61499: Design Elements

Fig. 5 shows an excerpt of the meta-model that describes the main elements of the IEC 61499 standard. In this meta-model, the core class is `IEC 61499` that contains a set of Function Blocks and events. Each `Function Block` contains a set of ports and it can be a `BasicFBType` or a generic `FBType`. The type `BasicFBType` may contain an `ECC` object. The objects of type `ECC` have a set of `ECCStates` and a reference to their initial state (reference `start`).

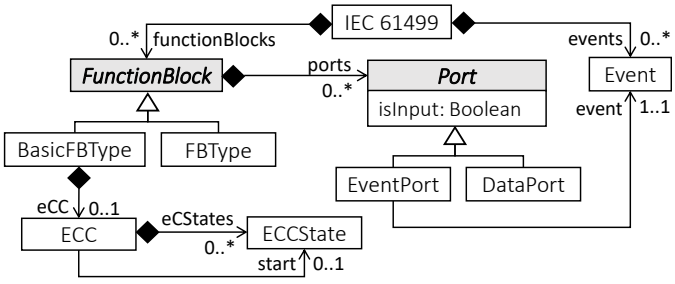

Fig. 5: IEC 61499 meta-model excerpt.

B. Interpreter: Runtime Elements

The interpreter was designed using a meta-model that captures the main properties and primitives of the language [5]. Fig. 6 shows the meta-model of our language called: *Operational Semantics for IEC 61499*. This language extends the base meta-model of IEC 61499 with the behavioral structure employing a dynamic meta-modeling approach [21]. Specifically, it is designed using a decorator pattern [22] to augment the language elements of IEC 61499 with the specification of the semantics that follows an event-based execution.

The `EventManager` is the core class of our Operational Semantics-language that contains the set of `Transactions`. Each transaction has one input event and may contain a set of output events. The class `EventOccurrence` represents the triggered events with two fields `ignored` and `active`. The attribute `ignored` tracks whether the event has been dismissed, and the `active` attribute stores whether

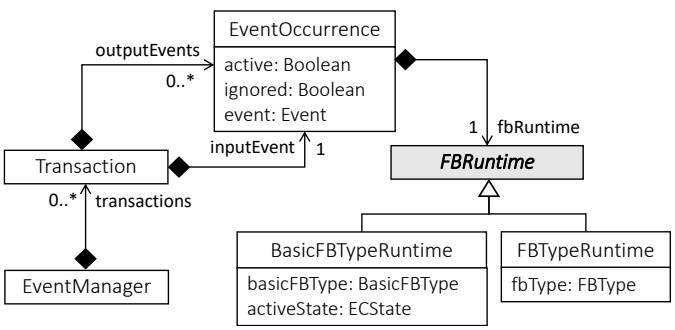

Fig. 6: Operational Semantics meta-model.

the event is still available for crossing a transition. Each `EventOccurrence` may contain one `FBRuntime`. The execution of different kinds of FBs varies, therefore, this class is abstract. `BasicFBTypeRuntime` and `FBTypeRuntime` extend this class to control the execution of BFBs and FBs, respectively.

C. Interpreter: Operational Semantics

Algorithm 1 shows the operational semantics, which is realized in the interpreter, semi-formally as pseudo-code. The entry point of this algorithm is the method `main` in the lines 1-6, which receives a sequence of transactions as an input. For each input transaction, the method `ProcessTransaction` (line 7) is called, which returns an FB. This FB will be assigned to a potential next transaction (line 4).

The method `ProcessTransaction` (lines 7-18) receives a `transaction` as a parameter. As a first step, this method initializes all variables within the FB (line 9). Secondly, it evaluates the transition condition of the `activeState` (line 12) by calling the method `EvalTransition`. Please note that transitions have to be evaluated in the order of their priority. Basically, the method `EvalTransition` (line 20) sorts all output transitions by priority (line 22), and then checks the output transitions sequentially (lines 23-37). In order to check the `ECTransitions`, this method implements the transition conditions described in Section III-C, i.e., 1-Condition (line 24), Event Condition (line 26), Guard Condition (line 31), and Combined Condition (line 34). After this check, a set of actions is performed if a valid transition is found (lines 13-17). In the method `consumeEvent`, the `EventOcurrence` is consumed by assigning `false` to the attributes `ignored` and `active`. Then, in line 15, the active state is changed to the target transition destination, and the entry actions of this state are executed. The `PerformEntryActions` method executes the algorithms of the `ECState` and adds the output events to the current transaction. Lastly, in line 16, the algorithm continues with the evaluation of the following outgoing transitions.

D. Integration with Service Sequences

Our interpreter can be used for evaluating FB types. Unit tests executed by an interpreter can provide early feedback to the developer. Hametner et al. [9] executed unit tests in a runtime after specifying them as Service models. Within the runtime, directly assigning a specific state of the FB or setting internal variables to a specific value violates the execution semantics of IEC 61499. Such unit tests therefore need to manually put an FB into the desired state (fixture), resulting in long and complex test sequences [9]. By applying our interpreter, the FB can be directly configured to a desired state before modifying the model based on the implemented semantics.

Service models describe a relation between input and output events of an FB in a number of Service Sequences. For BFBs, they thus visualize possible scenarios that are implemented in the ECC. A single Service Sequence captures a scenario at the interface, such as `initialization` or `normal operation`. As an example, we consider two Service Sequences for the BFB `E_CTU` (Fig. 7), which is a counter block and defined in the standard. A Service Sequence is comprised of one or more Service Transactions. The Service model has two interfaces: For specifying test cases, the interface showing the externally observable events is relevant.

Algorithm 1: Interpreter Execution Algorithm

```
   Input: Sequence of Transactions
1  Function Main:
2      forall transaction T_i within Transactions do
3          FB_i ← ProcessTransaction (T_i)
4          if Exist Next Transaction then
5              FB_{i+1} within Next Transaction ← FB_i
6
7  def ProcessTransaction(T):
8      FB ← Function Block within T.inputEvent
9      Initialize FB Variables ← {}
10     eCC ← ECC within FB
11     activeState ← Initialize Active State
12     Tr_f ← EvalTransition (T,activeState)
13     while Tr_f is not null do
14         T.inputEvent.active ← false
15         activeState ← Tr_f.destination
16         PerformEntryActions (activeState)
17         Tr_f ← EvalTransition (T, activeState)
18     return FB
19
20 def EvalTransition(T, activeState):
21     outTransitions ← Tr_i within activeState
22     Sort outTransitions by Priority
23     forall ECTransition Tr_i within outTransitions do
24         if Tr_i has 1-condition then
25             return Tr_i
26         if Tr_i has event condition then
27             event ← Tr_i.conditionEvent
28             if T.inputEvent is active and
                 T.inputEvent.equals(event) then
29                 T.inputEvent.ignored ← false
30                 return Tr_i
31         if Tr_i has guard condition then
32             if evaluate guard condition then
33                 return Tr_i
34         if Tr_i has combined condition then
35             if evaluate event and guard condition then
36                 T.inputEvent.ignored ← false
37                 return Tr_i
38     return null
39
40 def PerformEntryActions(T,ECState):
41     forall Action A_i within ECState do
42         if Exist A_i.algorithm then
43             execute A_i.algorithm
44         if Exist A_i.output event then
45             add A_i.output event to transaction T
```

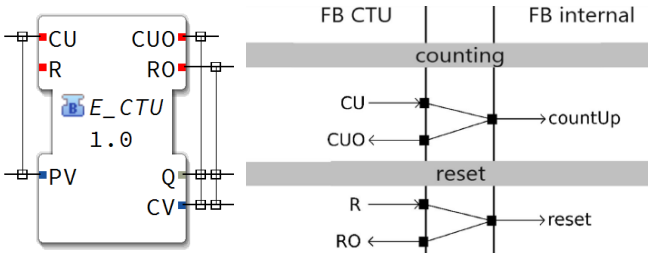

Fig. 7: Interface of the block E_CTU defined in IEC 61499 (left). It counts up upon receiving an event CU and outputs the counter value CV. The output Q compares CV and the input data (process value, PV). Two Service Sequences are illustrated that model the relationship between input and output events.

The scenario `counting` is started by an input event CU, the scenario `reset` by an input event R. They are InputPrimitives and start the corresponding Service Transaction. The resulting output events internally and externally are connected (OutputPrimitives). Internally, the algorithms `countUp` and `reset` are triggered. Externally, the respective output event CUO or R are then sent. Associated data is modeled as a parameter of the respective primitive [8].

When using the Service model as test specification, we need to identify the equivalencies between the model and the execution framework (see Table I). Each Service Sequence is used as a test case with an own EventManager. A Service Transaction corresponds to a Transaction of the EventManager. For the event specified in the InputPrimitive, an EventOccurrence is generated, and set as the input event of the Transaction. When processing the EventManager, the BFB is triggered with these input events one after another. The generated output events of a certain input event are stored in the respective Transaction. After execution, they can be compared to the expected events specified in the OutputPrimitives of the corresponding Service Transaction in the model.

TABLE I: Mapping between Service model and execution framework

Service model	Operational semantics
Service Sequence	Test case with EventManager
Service Transaction	Transaction
Input Primitive (event)	Input EventOccurrence
Input Primitive (parameter)	Data values at the input
Output Primitive (event)	Expected output EventOccurrence
Output Primitive (parameter)	Data values at the output
Parameter	Pre-defined/expected data value
Internal event	- (not applicable)

E. Tool Support

The interpreter was developed as an Eclipse plug-in which extends the capabilities of Eclipse 4diac. The language of the interpreter was implemented using Ecore, which is the meta-modeling language provided by the Eclipse Modeling Framework [23]. In this sense, the *Operational Semantics* meta-model is an extended version of the current meta-model for the language IEC 61499 that forms the basis of the 4diac IDE. The code is part of the public Eclipse repository[3].

A dedicated environment for testing FBs is integrated in the type editor of Eclipse 4diac, from where the interpreter can be started. The tests involve sending events to the respective FB inputs, hence, a list with EventOccurrences has to be prepared. Then, a `BasicFBTypeRuntime` with the `InputQueue` is created, which contains a reference to the FB under test. When the run()-method is called, all EventOccurrences from the InputQueue are processed.

Our framework aims at testing user-defined orchestrator FBs of hierarchical application designs. The interpreter can execute simple algorithms that are provided in ST. Variable assignments are typically done in ST, as they cannot be represented in the ECC directly, unlike loops and conditions.

V. Evaluation Setup

We evaluated our implementation using (i) the Basic FBs defined in the standard, and (ii) the demonstration example that we presented in Section III. For each FB, we compare the results of our interpreter to those obtained from the runtime 4diac FORTE. First, we defined test sequences for each BFB that cover all states and transitions of the ECC. We ensured that each transition condition evaluates both to `true` and to `false` during the execution. Next, we manually recorded the output data and event occurrences from 4diac FORTE, i.e., launched a runtime on our PC and accessed it via the *FBTester*-module of 4diac IDE. For each FB type, we manually executed our defined test sequences: we adjusted the input data and triggered events. Observing the outputs allowed us to record the expected output events and data per input event. Finally, we created a Service model that captured these test sequences. Using our unit test environment in Java, we compared the results of our interpreter to those obtained from 4diac FORTE.

A. Basic FBs defined in IEC 61499

TABLE II: Basic FBs defined in IEC 61499. Listing relevant transition conditions and algorithm statements in Structured Text (ST).

Block Types	Transition Conditions	ST Statements
various	1	
CTD, CTU	R	
CTD	CD[CV >= 1]	CV := PV; Q := (CV <= 0);
		CV := CV – 1;
CTU	CU[CV < 65535]	CV := 0; Q := FALSE; CV := CV + 1;
		Q := (CV >= PV);
D_FF	CLK[D]	Q := D
	CLK[NOT D]	
MERGE, REND	E1, E2	
PERMIT	EI[PERMIT]	
RS, SR	R, S	
SELECT,	EI0[NOT G]	
SWITCH	EI1[G]	
SPLIT	EI	
T_FF	CLK	Q := NOT Q;
TABLE_CTRL	[N = 0], [N > 0]	
	INIT	DTO := DT[0];
	CLK[CV<MIN(3,N–1)]	DTO := DT[CV];

The interfaces and ECCs of 12 Basic FBs are defined in Annex A of the standard IEC 61499-1. Table II provides an overview of their transition conditions and algorithms. They include all variants of condition expressions. The algorithms are comprised of assignment statements, where a value is calculated and assigned to a variable, i.e., an output pin or

[3]4diac IDE Repository: git.eclipse.org/c/4diac/org.eclipse.4diac.ide.git

an internal variable of the FB. Conditions or loops can be implemented either in the graphical ECC model or in the embedded ST algorithm.

We demonstrate our approach with an example test case from [9] for the block E_CTU. A Service model for this FB was presented in Fig. 7, which comprises the Service Sequences counting and resetting. Each row in Table III corresponds to a Service Transaction with one input primitive and n output primitives. The inputs are comprised of one event (`CU` or `R`) and a data value (`PV`). We recorded the expected output events and data for each input configuration from 4diac FORTE. They are comprised of two events, `CUO` and `RO`, and the data outputs `Q` and `CV`. As shown in the Service Sequences (cf. Fig. 7), only a single output event is received.

TABLE III: Test Case for the counter block E_CTU [9]

Service Sequence	Inputs			Outputs			
	CU	R	PV	CUO	RO	Q	CV
reset		x	0		x	FALSE	0
counting	x		0	x		TRUE	1
reset		x	0		x	FALSE	0
counting	x		1	x		TRUE	1

We implemented all test cases as a Service model to comply with the standard IEC 61499.

1) Implementing test cases in 4diac IDE: A graphical editor for Service models is available in 4diac IDE. The models can be stored in the standardized XML-format.

2) Implementing test cases in Java: The Service Sequences can be also created manually using generated EMF code. The FB type is loaded into the execution framework. Using our predefined methods, we can (i) create an empty Service model with a Service Sequence, and (ii) add Service Transactions to a Service Sequence `seq`. For our test case of E_CTU, we thus add four transactions:

```
1 TestService.Builder(seq).transactions(
2       new ServiceTransaction("R", "RO", "Q:=FALSE; CV:=0;"),
3       new ServiceTransaction("CU", "CUO", "Q:=TRUE; CV:=1;"),
4       new ServiceTransaction("R", "RO", "Q:=FALSE; CV:=0;"),
5       new ServiceTransaction("CU", "CUO", "Q:=TRUE; CV:=1;")
6 ).build();
```

Each `Transaction` encapsulates the data of a row from our Table III. The method `add` builds the required model. The output parameters are separated by a semicolon. As each input event triggers only a single output event, only one output primitive is required, but also lists of output events are supported. The interpreter for a FB type `fb` is launched via `run(fb, seq, state)`, where `state` denotes the name of the state that is initially active. The `run`-method creates a new FBTypeRuntime with the respective active state, and an EventManager. For each ServiceTransaction, a new Transaction is created in the EventManager. Finally, the results from the execution (as stored in each Transaction) are compared to the expected results (as stored in the ServiceTransactions).

We performed these steps for all FBs that are listed in Table II. The results from our execution framework match those of 4diac IDE.

B. Orchestrator FB of the running example

Testing user-defined Basic FBs is also feasible. We created three exemplary Service Sequences for the Basic FB `StationCtrl` (Fig. 3) from our running example in Section III. These sequences show possible paths through the ECC (cf. Fig. 4) for the scenarios of (1) initialization, (2) normal operation, and (3) erroneous behavior of the conveyor belt. We created a Service model in Fig. 8 that serves as a test specification. Each scenario is converted to one test case. We illustrate the input data at the time of the scenario as a parameter (`ErrorCode:=0`).

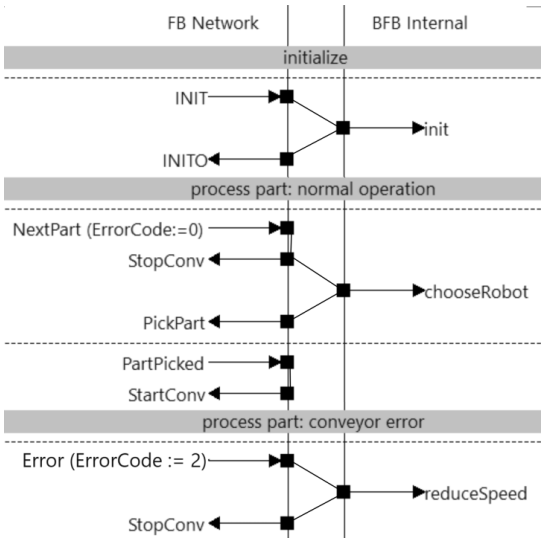

Fig. 8: Service Sequence for the orchestrator StationCtrl of our running example. It shows three possible scenarios of the ECC in Fig. 4: (1) initialization, (2) normal operation for processing a part, and (3) error-handling for unexpected behavior of the conveyor belt.

The scenario normal operation is comprised of two transactions. Note that for the first transaction, a single input event (`NextPart`) causes two output events, `StopConv` and `PickPart`. The Service model cannot capture the initial state of a sequence. While scenarios 1 and 2 begin at the `START` state of the ECC, Scenario 3 models an error case and its initial state is `processingPart`. In our execution framework, we have to adjust the activeState of the BasicFBTypeRuntime:

```
1       run(fb, seq3, "processingPart");
```

VI. Evaluation Results and Discussion

Our interpreter fully implements the execution semantics of ECCs. The transition conditions within the ECC are specified in the general-purpose language ST. Currently, all types of transition conditions (cf. Section 3) are supported, but the ST expressions are limited to common data types and common operations. Our implementation supports variable assignments in ST and conditions, but does not cover function calls, as used by the block E_TABLE_CTRL. Furthermore, commonly used elementary data types are supported, but neither arrays nor compound data types. Such advanced features will need to be added to the ST interpreter in the future. Therefore, our

execution framework can interpret the Basic FB types defined in IEC 61499 except for the type TABLE_CTRL.

Concerning unit-tests of FBs, we created a test framework that allows specifying tests both as a model and as Java code. We implemented algorithms that automatically compare the results from the FBTypeRuntime with the defined test cases and provide helpful feature to (i) define a start state for the execution, and to (ii) set defined values for variables of the FB. These features allow providing fixtures for the test execution and are particularly useful for testing the correct function of error handling. Compared to prior approaches that execute FBs directly on the runtime [9], our execution framework can provide test fixtures that involve a specified start state. In [9], it was reported that testing error handling requires long and complex scenarios and is not feasible with their approach.

VII. Conclusion and Future Work

We provided a detailed description of the execution semantics for IEC 61499 Basic FBs, both in natural language and as pseudo code. Our implementation in Java and Xtend can interpret Basic FBs, as are typically created for orchestrating application parts of a station. As a sample application for our interpreter, we demonstrated its utility for testing FBs. Furthermore, it may facilitate:

- analyzing models, as the semantics is integrated in the metamodel,
- visualizing scenarios based on an implemented ECC and persisting them in an IEC 61499-compliant format,
- identifying inconsistencies between several views of the Basic FB model (between ECC and Service model),
- validating refactoring results, as the black-box behavior of the FB before and after the applied refactoring must remain unchanged,
- validating code generators, as the results of the model execution can be compared to the execution of the generated FB in the runtime.

Our approach furthermore demonstrates that DSMLs for industrial purposes can benefit from technologies from the DSML engineering domain. Current development methodologies in industrial automation do not fully utilize the available practices from DSML engineering. Language workbenches can further improve the support for industrial DSMLs. For instance, the language workbench GEMOC provides support for debugging DSMLs and recording execution traces [24].

In future work, we will extend our interpreter to cover not only individual FBs, but also networks of FBs. Furthermore, we intend to show its practicality also for other approaches than unit-testing FBs and apply further well-known practices from DSML engineering.

Acknowledgements

Work partially funded by the Austrian Science Fund (P 30525-N31) and the European Union's 1-SWARM project under grant agreement 871743.

References

[1] H. Yin and H. Hansson, "Fighting CPS Complexity by Component-Based Software Development of Multi-Mode Systems," *Designs*, vol. 2, no. 4, p. 39, 2018.

[2] M. Törngren and P. Grogan, "How to Deal with the Complexity of Future Cyber-Physical Systems?," *Designs*, vol. 2, no. 4, p. 40, 2018.

[3] R. Harrison, D. Vera, and B. Ahmad, "Engineering Methods and Tools for Cyber–Physical Automation Systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 973–985, 2016.

[4] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

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

[6] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.

[7] IEC TC65/WG6, "IEC 61499-1, Function Blocks - part 1: Architecture v2.0: Edition 2.0."

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

[9] R. Hametner, I. Hegny, and A. Zoitl, "A unit-test framework for event-driven control components modeled in IEC 61499," in *Int. Conf. Emerging Technology and Factory Automation (ETFA)*, pp. 1–8, 2014.

[10] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[11] M. L. Alvarez, I. Sarachaga, A. Burgos, E. Estévez, and M. Marcos, "A methodological approach to model-driven design and development of automation systems," *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 1, pp. 67–79, 2016.

[12] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014.

[13] Object Management Group, "Systems Modeling Language (OMG SysML): Version 1.6," November 2019.

[14] M. Fowler and R. Parsons, *Domain-specific Languages*. Addison-Wesley signature series, Addison-Wesley, 2011.

[15] Eclipse 4diac, "Eclipse 4diac - The Open Source Environment for Distributed Industrial Automation and Control Systems." https://www.eclipse.org/4diac, 2019.

[16] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Test case generation approach for industrial automation systems," in *Int. Conf. on Automation, Robotics and Applications (ICARA)*, pp. 57–62, IEEE, 2011.

[17] R. Hametner, D. Winkler, T. Östreicher, N. Surnic, and S. Biffl, "Selecting UML models for test-driven development along the automation systems engineering process," in *Int. Conf. Emerging Technology and Factory Automation (ETFA)*, pp. 1–4, IEEE, 2010.

[18] Object Management Group, "Unified modelling language: 2.5."

[19] R. Sinha, P. S. Roop, G. Shaw, Z. Salcic, and M. M. Y. Kuo, "Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 59–68, 2016.

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

[21] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer, "Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML," in *Int. Conf. on the Unified Modeling Language*, pp. 323–337, Springer, 2000.

[22] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[23] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2008.

[24] D. Leroy, E. Bousse, M. Wimmer, T. Mayerhofer, B. Combemale, and W. Schwinger, "Behavioral interfaces for executable DSLs," *Software & Systems Modeling*, vol. 19, no. 4, pp. 1015–1043, 2020.