Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.
Digital Object Identifier XX.XXXX/YYY.20XX.ZZZZZZZ

Develop Once, Test Everywhere: Cross-Platform Development of Distributed Control Software

BIANCA WIESMAYR[1], MELANIE WINTER[1], MIDHUN XAVIER[2], SANDEEP PATIL[2], VALERIY VYATKIN[2,4] (Fellow, IEEE), AND ALOIS ZOITL[3] (Senior Member, IEEE)

[1]LIT CPS Lab, Johannes Kepler University Linz, Austria (e-mail: bianca.wiesmayr@jku.at)
[2]Luleå University of Technology, Luleå, Sweden (e-mails: midhun.xavier@ltu.se, sandeep.patil@ltu.se)
[3]CDL VaSiCS, LIT CPS Lab, Johannes Kepler University Linz, Austria (e-mail: alois.zoitl@jku.at)
[4]Aalto University, Helsinki, Finland (e-mail: Valeriy.Vyatkin@aalto.fi)

Corresponding author: Valeriy Vyatkin (e-mail: Valeriy.Vyatkin@aalto.fi).

This work was sponsored in part by the Horizon Europe project Zero-SWARM funded by the European Commission (grant agreement: 101057083).

ABSTRACT IEC 61499 is an executable event-based language for control software that allows visual and textual implementation of individual software components, so-called Function Blocks (FBs). In heterogeneous environments, relevant features are the transfer of software and libraries between tools (portability) and the configuration of multiple runtime environments from a single IDE (configurability). Due to ambiguities and deviations from the standard, the FB behaviour slightly varies on each available platform. Such variations are not only observed in IEC 61499-based software and limit portability in practice. This paper therefore investigates an approach for software testing which involves generating portable test code from a specification model. For cross-platform validation, the test code is executed in each relevant run-time environment. We have evaluated our approach in a test-driven development process for a drilling demonstrator. Eclipse 4diac was extended to automatically generate the test code, which was subsequently ported to another IDE. The generation mechanisms consider platform-specific deviations from the IEC 61499-standard to promote portability. We demonstrated the feasibility of cross-platform testing for IEC 61499-based components, enabling further work in the area of test case generation.

INDEX TERMS Model-based Testing, IEC 61499, Function Blocks, Service Sequences, Portability

I. INTRODUCTION

Realising flexible industrial automation systems requires an approach for autonomous and distributed designs [1]. Programmable Logic Controllers (PLCs) are the established platform for real-time control software that accesses sensors and actuators [2]. Standards play an important role in distributed automation, for instance, IEC 61131-3 and IEC 61499, which define programming paradigms for control software development [1]. Multiple interacting PLCs form a distributed control system. Providing the respective engineering methodologies and models is a goal of IEC 61499 [3]. Heterogeneous systems can even be composed of PLCs from various vendors and programmed with different tools [4] [5]. Furthermore, a single development tool can distribute control code across multiple runtime environments (RTEs) [6], motivating the need to execute component tests in each of these RTEs. Developers of IEC 61499 library modules [7] will also need to provide their modules to users of various development environments. Despite the focus on portability [3] and the standardized XML format for data exchange [8], IEC 61499-based software components must often be modified during the porting process [9], [10]. Due to varying execution behaviour, the ported software may behave differently on each platform [10], [11], possibly leading to malfunctions of the distributed control system. Therefore, it is crucial to thoroughly test an IEC 61499 application on each relevant target platform before using the software in a real-world system. A platform-independent test specification has the potential to greatly reduce the involved effort.

Evaluating a system's correctness typically involves providing test data and observing the system's reaction. Such tests are specified manually, obtained from models, or generated using other techniques [12]. The engineering processes of complex control systems additionally require that PLC platforms ensure reliability through simulation and verification, rather than relying on iterative testing of a cyber-physical system [2]. Simulation methods evaluate only a system model and, thus, can provide results faster [12]. In

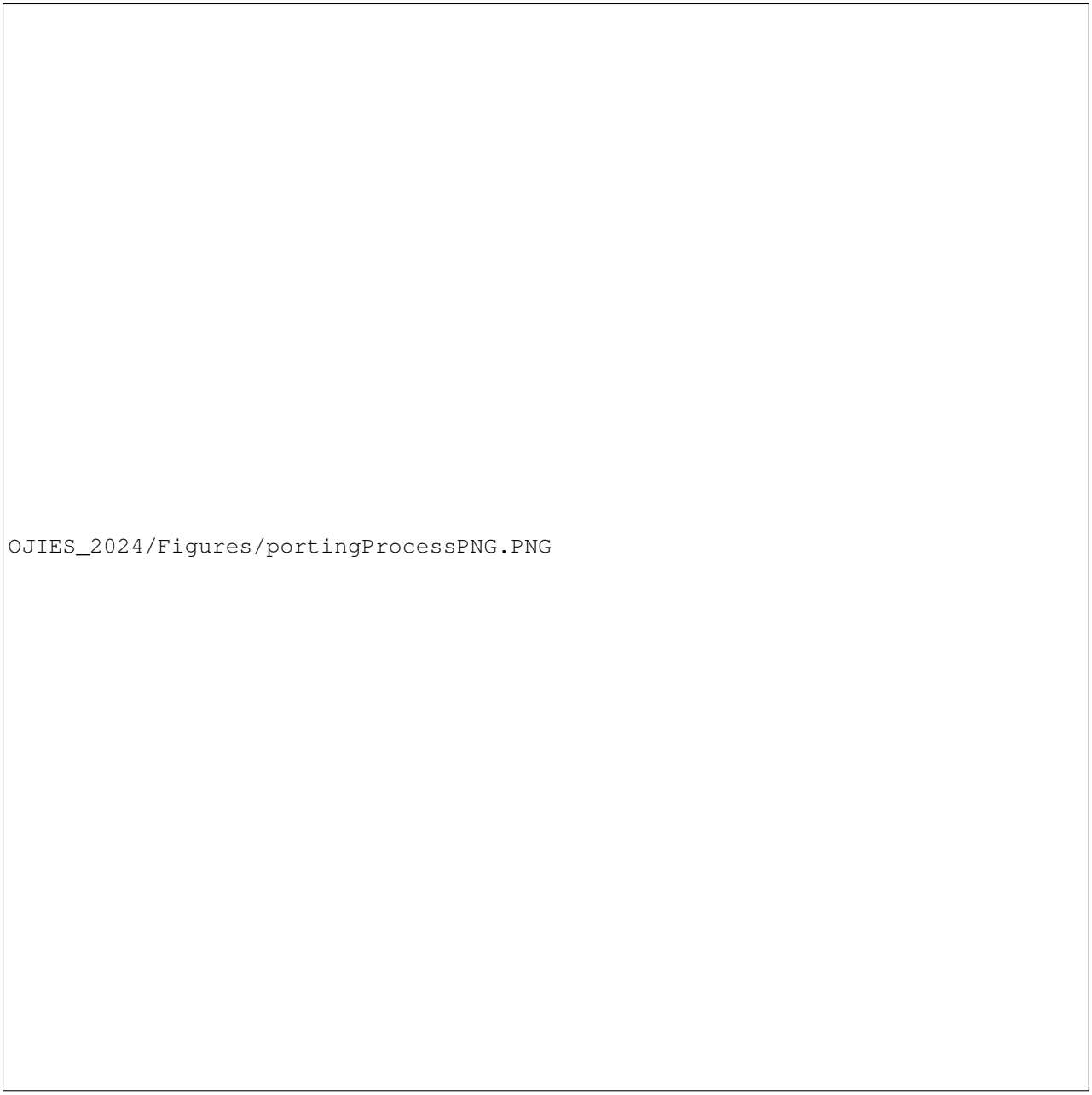


FIGURE 1. Process of porting test code for IEC 61499 Function Blocks between execution environments.

the context of IEC 61499, executing test cases within a runtime environment can be considered a simulation. A certain degree of test automation is required to efficiently evaluate this software. Existing test processes for IEC 61499-based software rely on a framework in the target platform (e.g., [13]) and are therefore not applicable for cross-platform development of IEC 61499-based software. When (re-)using parts of IEC 61499 software in multiple platforms, extensions to the IEC 61499 standard may not be available in all platforms (equally). Hence, we are investigating an approach for test automation that does not require any extensions of IEC 61499.

Research Question (RQ)

How can we automate the testing of IEC 61499 software components to evaluate their correct behaviour across platforms from different vendors?

This paper aims to address the challenge of evaluating the effect of porting software components to other platforms or configuring additional RTEs. The test framework allows us to execute tests in any IEC 61499-compliant RTE. Based on our initial concept presented in [14], we generate an IEC 61499-compliant test application that automatically provides test events and data, compares the results of the software com-

ponent under test with the expected observable behaviour and summarises the results so that they are accessible by the user. The general process is visualised in Figure 1. Like Hametner et al. [13], we use service sequence models as test specifications. Related work in the context of testing and porting IEC 61499 components is outlined in Section II. Section III introduces a running example. Based on this example, Section IV outlines the envisioned methodology to test FBs on any IEC 61499-compliant platform. One of these IDEs, the 4diac IDE from the Eclipse 4diac open source project, was extended to generate the test application. The generation rules for a test application and their implementation are described in Section V. Realised as a composite FB, the test application is portable across various IEC 61499 platforms and enables validation of the correct functionality before deployment in real-world machinery. We evaluated our approach using a demonstrator (Section V-A). Section VII lists the identified portability issues and programming errors that we could detect using our test suite. In addition, we discuss limitations of our approach before concluding our paper in Section IX.

II. RELATED WORK

Portability, interoperability, configurability, and distribution across devices are key goals of IEC 61499 [15]–[17]. This has led to its application in use cases of flexible manufacturing systems, such as multi-agent systems [18], [19]. However, portability between vendors has not yet been achieved due to vendor-specific execution behaviour [20]. In this section, we therefore review recent studies on porting software between engineering platforms. Furthermore, techniques for improving the reliability of control software are discussed with a focus on testing approaches. The approach proposed in this paper aims to test software components on different platforms, thus, promoting reusability of software as well as supporting the process of porting software components.

A. PORTABILITY OF CONTROL SOFTWARE

Both standards that define programming languages for control software, i.e., IEC 61131-3 [21] and IEC 61499-1 [3], define XML formats for exchanging software between PLC environments [8], [10], [22], [23]. The standardised format contributes to code exchange between tools from different vendors [23]. As existing tools only support a varying subset of features, portability is limited in practice [10]. Certain vendors also use custom XML tags to store additional information in their projects which are not covered by the standard itself, such as namespaces in IEC 61499. The resulting software may still be portable if the parser in the target IDE discards unknown XML tags that carry additional information [9]. The language specification focusses on control software without detailing applied communication standards or visualisation options. Language extensions, such as modelling elements for communication [24], are not widely supported and therefore not yet portable. One of the IDEs allows specifying the human-machine interaction (HMI) as part of the control code with dedicated blocks, so-called CAT elements [9]. Converter programs can help translate programs between syntactic variants of IEC 61499 [9], but vendor-specific extensions can typically not be transferred. Even syntactic equivalence does not guarantee portability. Nowadays, cross-platform development does not require porting because a single IDE allows deploying software to multiple run-time environments [25]. The IEC 61499 language semantics is not specified formally and is subject to interpretation. Different execution semantics have emerged, which can affect the behaviour of the cyber-physical system [26]. Hence, migrating IEC 61499-based software to different run-time environments can introduce errors that may cause damage to operators or equipment. While the syntactic portability has been addressed [9], [10], different execution behaviours cannot yet be detected (semi-)automatically. It is therefore crucial to thoroughly test IEC 61499 applications in the target platform before deployment. Existing testing mechanisms enable systematically evaluating the implemented execution semantics of IEC 61499 runtime environments [10], [11], [27], but a solution for porting test cases including their execution framework to different platforms is not yet available. We have presented our initial concept in [14].

B. VALIDATING CONTROL SOFTWARE

Cengic and Akesson [26] followed the approach of creating a formal semantics specification for each runtime environment. Subsequently, formal verification methods can be applied to identify errors in the code [28], which can enhance a system's reliability by checking various properties. Formal verification does not require access to any RTE, but a model of the execution behaviour is required. Sinha et al. [12] provide an overview of formal methods for IEC 61499. Formal verification may uncover errors that do not occur during simulations, thus, identifying certain undesirable situations [29]. Control software has to be continuously adapted to changing requirements. Research has therefore provided guidance on the correct evolution of control software developed in IEC 61499 [30]. Verification and testing can complement each other [31]. During development, tests provide early feedback, even if the model is still incomplete. Furthermore, errors that are introduced during the deployment may lead to issues encountered during runtime, but might not be revealed by formal methods [32] [33]. Hence, testing deployed software directly in the run-time environment remains valuable.

Developers can apply various testing strategies, which we differentiate based on the involved software activities (e.g., unit tests or integration tests), the maturity of the software, and the degree of automation [34]. Unit testing is a fundamental approach to software testing, which evaluates the implementation of a piece of software [34] to ensure software reliability. In IEC 61499, the relevant units are individual Function Blocks (FBs) [13]. Executing a test requires providing event and data signals. For control engineers who develop FBs, it can be challenging to manually create a test FB and the required test application, which derives and collects the

test results. Model-based testing can reduce manual effort and also supports a "test first and fail" methodology, known as Test-Driven Development (TDD) [13], which is used in agile software engineering. After developing the control program for the entire system, functional tests can be conducted. This involves evaluating the control system by providing input data and verifying the output against expected results.

Tools should support engineers in specifying test cases to reduce the required software engineering knowledge and increase efficiency [13]. Model-based testing involves automating at least part of the testing activities. For IEC 61499 FBs, service sequences are suitable for specifying tests [13]. A test runner can execute these tests in an RTE and automatically evaluate the results [13]. Additionally, executing models directly can allow feedback without involving any RTE and is also feasible for service sequences [35]. The former approach requires specific tool support for a certain RTE, the latter cannot provide feedback regarding issues introduced in the deployment to an RTE. Our approach builds upon these works. As an alternative to service sequences, UML models have been used as test specifications [31]. From a state-based model, test cases can be derived using coverage-driven algorithms [31]. Using an evolutionary algorithm, test cases with a high coverage were generated directly from the FB model in [36]. Test case generation can augment our approach, which focusses on executing tests of any source on multiple platforms. Additional tool support would be required to use other test specifications than service sequences.

Two major problems are still associated with developing distributed control software that spans multiple platforms:

- The lack of automated tool support for RTE comparison makes comparing the behaviour and performance of FBs across different RTEs a challenging task. Currently, manual comparison is time-consuming and error-prone. Dedicated tools should analyse and evaluate the behaviour of FBs in different RTEs to ensure an accurate comparison.
- Software development for different RTEs is challenging because the compatibility and portability of an FB across different RTEs cannot be assumed.

For example, if an FB is initially developed and tested on one RTE, such as SE EcoRT, there might be a need to reuse that FB in another project in a different RTE, such as 4diac FORTE. Differences in RTE behaviour, programming languages, and underlying architectures can cause compatibility issues and hinder the seamless transfer of FBs between different RTEs.

C. CROSS-PLATFORM TESTS OF CONTROL SOFTWARE

Two main strategies are relevant for IEC 61499-based software. Firstly, **manually created test FBs** [10] can reveal the behaviour implemented in an RTE. Each test FB encompasses multiple test scenarios and embeds control logic. To indicate whether a test was successful, the expected result is compared with the result obtained from executing the control logic. The tests are implemented as a Basic FB with event and data pins. Each input event represents a test scenario linked to specific data inputs, while output events indicate the expected result and corresponding data outputs. When a test scenario is triggered, the state diagram (i.e., Execution Control Chart, ECC) executes an algorithm that assigns input values, generates outputs based on those values, and triggers the output event. The main purpose of these FBs was to identify differences between the execution behaviour in RTEs, not to test FB libraries. Similarly, small networks of FBs can further expand these test suites [11], [27]. As a second approach, **generating test code from a high-level test specification** has the potential to enable testing FB implementations [14].

III. RUNNING EXAMPLE: SIMPLE CALCULATION FB

As a running example, we use an FB that performs a simple calculation (Figure 2) based on the inputs according to the formula `DO1:=DI1+2*DI2`. It contains the following language elements:

- Input event `REQ` triggers the calculation.
- Data inputs `DI1`, `DI2` receive values from other FBs and are used in the algorithm.
- The data output `DO1` stores the result of the algorithm.
- The output event `CNF` is issued to indicate that the algorithm was completed and that the output result is ready to be used by other FBs.
- The algorithm `REQ` within the FB is executed whenever the input event `REQ` occurs. The algorithm assigns the result of the formula to the output variable DO1.

In our example, the FB performs the calculation if the values of `DI1` and `DI2` are between 1 and 1000 (cf. implementation of the state diagram in Figure 2). When triggering the `REQ` event with appropriate input values, the FB executes the algorithm and produces the respective output. Both cases

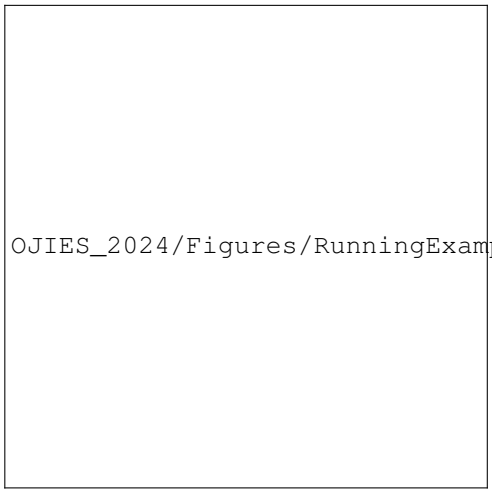


FIGURE 2. Running Example: FB performing simple calculation. FB interface defining the component, implementation as state diagram, and two usage scenarios modelled as service sequences.

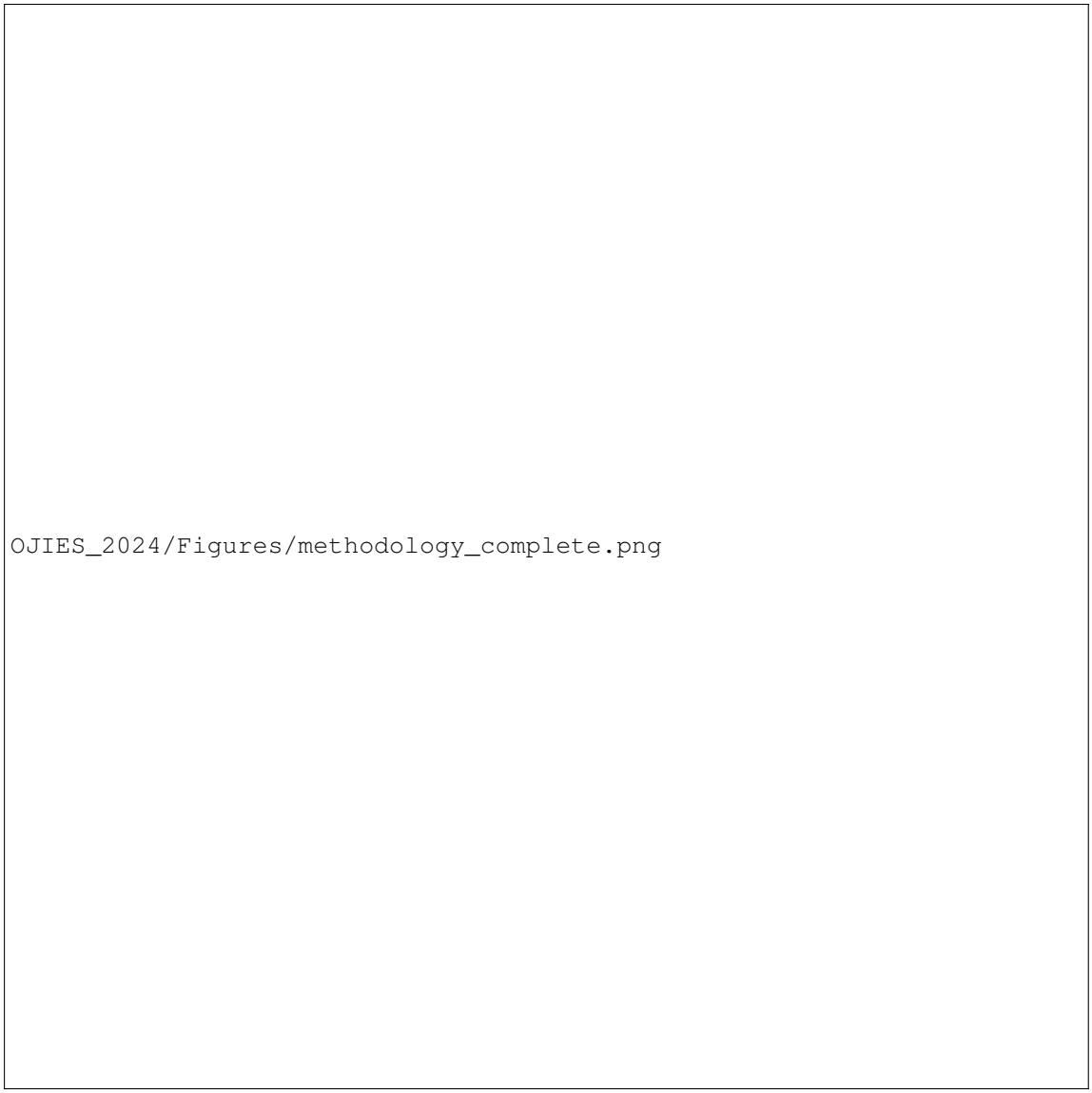


FIGURE 3. Overview of process for testing FBs across various software platforms. The seven-step process is partly automated with tools (blue boxes), partly it requires tool-assisted manual development. The test application can be ported to tools of various vendors and validate the expected behaviour of an FB directly in the target RTE, thus, possibly identifying differences in the execution semantics that affect the FB under test.

are modelled as service sequences and serve as test scenarios for the running example.

IV. METHODOLOGY FOR TESTING FBS

Our proposed approach for cross-platform FB testing is suitable for a test-driven development process, as well as for testing existing implementations. It involves specifying tests, executing tests within an IDE, generating the portable test application, as well as executing these tests on all relevant RTEs. A test application that is compliant with IEC 61499 can be ported to RTEs of various vendors. The approach is visualised in Figure 3 as a step-by-step approach using the generated test cases of our running example (Section III). In the following, we describe each step of the process in detail. We describe the envisioned development process based on our running example.

A. TEST-DRIVEN DEVELOPMENT OF FBS

The following steps describe a test-driven development process. For testing existing FB libraries, the process starts directly at step 4, as the implementation is already complete. In this case, test cases will be generated semi-automatically from the implementation to reduce the effort.

1) Creating a new Function Block type (FBT)

Reusable functionality should be encapsulated in an FB. This involves specifying the input/output events and data inputs/outputs. For the running example, this involves defining the interface of the calculation FB.

2) Specify test cases as service sequence models

Specification models are defined. In the running example, a service model with two sequences is provided to define the test scenarios for the FB (cf. Figure 3, step 2). The service model specifies the expected event occurrences, as well as the input values (`DI1` and `DI2`) and the expected output value (`DO1`) for each test case. The scenario of `test1` is triggered upon arrival of an event at the input `REQ`. The purpose of `test1` is to describe the FB behaviour by checking whether it correctly returns 19 when given input values of 5 and 7. Additionally, `test2` aims to evaluate the FB behaviour for an edge case, as one value will be out of range (i.e., `DI2:=INT#1001`). We expect that no addition is performed, and no output events are sent (cf. second sequence in step 2). Where output data values are expected, they are specified together with the output event.

3) Implement desired functionality of the FB type

When following a TDD process, the functionality of the FB is implemented at this stage. The specified tests can be used for iteratively evaluating the correctness. For instance, the implemented behavior of an FB can be analysed using model interpreters to receive rapid feedback on any changes. For the running example, we assume that the limit check of DI2 was not yet implemented. This scenario is illustrated in step 3 of Figure 3. When the test cases are executed using a model interpreter [35], the feedback shows that an unexpected event occurrence (i.e., CNF) has been output by the FB under test. By comparing the actual output with the expected output for each test case, the implemented FB behaviour is evaluated automatically. As a result, the transition condition of the state diagram can be updated to include the missing check for `DI2<=1000`. Afterwards, evaluating the service sequence is successful.

B. TESTING IMPLEMENTED FBS

After an FB has been developed, it needs to be evaluated in an RTE. While manual testing is feasible on all existing RTEs, automating the process allows to reduce the development time and effort. Hence, the following steps describe the process of model-based testing by generating an IEC 61499-based test application from the test specification.

4) Defining (additional) test cases for the implemented FB

Especially when validating the behavior of an FB in different platforms, creating test cases for additional corner cases may be useful. Even when a test-driven development approach is followed, a comprehensive test suite may not be available. Using a model interpreter and its accompanying execution framework [35] allows recording additional test cases based on specified events and/or data inputs. This also allows adding test suites to existing implementations. Step 4 in Figure 3 shows the dialogue for recording a service sequence where DI1 is out of range. The resulting graphical diagram is added to the FB type specification and is shown on the right. Developers have to manually check whether the result matches the expected behaviour of the developed FB. Recorded tests can serve as regression tests, as they capture the actual behaviour of an executed FB. This can ensure that an FB is evaluated comprehensively during model evolution.

5) Generate test application for specified test cases

Once the implementation has been completed, the correct real-time behavior of an FB has to be evaluated in a run-time environment. The test application ensures that developers do not need to manually interact with an FB and observe the outputs. Hence, various components are required for issuing test signals, which can be automatically generated. They are visualized in step 5 of Figure 3 and described as follows:

- **(A) Test signal generator:** This FB generates the input signals based on the service model and supplies the required events and data to the FB under test (REQ, DI1, DI2). It also notifies the test application of the expected output events (CNF or none). Additionally, it provides the expected output values (DO1), which are set in algorithms.
- **(B) Matcher:** This FB compares the execution results of the FB under test with the expected results that are provided by the test signal generator.
- **(C) Multiplexer:** The next FB forwards the result (ERROR, SUCCESS) of each test case (i.e., service sequence) to the output, together with the name of the executed test case. This helps developers to identify the faulty test case if there are any problems.
- **(D) Test application composite:** A composite FB encapsulate the test application so that it can be easily deployed to an RTE. All components are interconnected to provide a simple interface. Developers can run each test case by triggering the respective events. An additional event pin "run_all" is provided to execute all test cases sequentially based on a single event trigger. This functionality is handled by an additional FB which initiates these further test cases.

Some of the components (e.g., the matcher) also require a timer to wait for the results of the FB under test, thus, ensuring that no unexpected event outputs are detected. As a result, timer FBs are included in the test application. Note that the test signal generator, the FB under test, and the matcher are instantiated once per test case. This ensures that the internal state of these components does not affect further test cases. The FBs are guaranteed to initiate the execution from the START-state. As all FBs for the test application require an internal state, they are realized as Basic FBs. Although Figure 3 visualizes the test environment for a Basic FB (i.e., the running example), any kind of FB can serve as the FB under test. Only the interface definition and the service model are required. This also means that application parts can be tested as long as they are integrated into an FB.

Based on the results provided in [10], [14], we derived transformation rules for creating test code from service models. We implemented these rules in Eclipse 4diac [6], which provides an open source IDE for IEC 61499-based software.

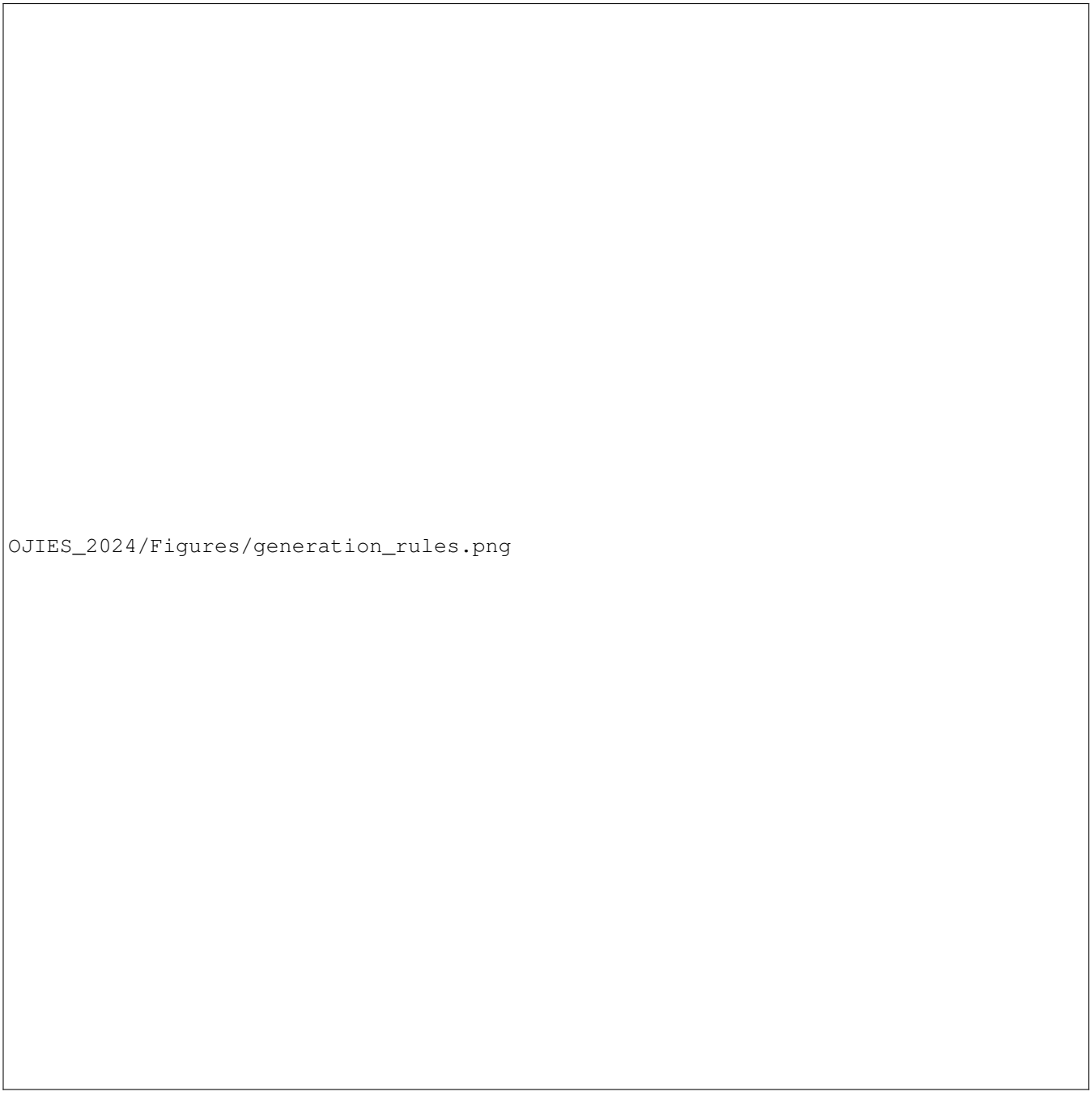


FIGURE 4. Test application generated for two service sequences of the running example. Relevant regions are highlighted including their relation to the service model.

- A **service model** serves as the test suite and includes one or more service sequences. A single test application is generated for the whole service model.
- Each **service sequence** serves as one test case. We need one event input per test case in the test application FB, which will trigger the execution of this test. We include its name in the event pin to relate the parts of the test application with the respective service sequences. Once the service sequence was completed, an ERROR or SUCCESS event is issued, together with the name of the service sequence. A service sequence can consist of several service transactions, which define the flow of events along the sequence.
- Each **service transaction** is comprised of an input primitive (ingoing arrow) and any number of output primitives (outgoing arrows). The service transactions are processed one after another. An event issued by the matcher (nextCase) indicates that a transaction has been completed.
- The **input primitive** describes the event that initiates a

transaction. The test signal generator FB has to issue the input event specified in a transaction. They are supplied to the FB under test via the respective connections together with the associated data. As a result, the test signal generator FB requires one output pin (events and data) for each input pin of the FB under test.

- The **output primitives** describe the event(s) that is/are caused by the input event. The test signal generator FB issues an event indicating all output events that are part of a test sequence. They are supplied to the FB under test via the respective connections together with the associated data. As a result, the test signal generator FB requires one output pin (events and data) for each input pin of the FB under test. The matcher FB receives information about the expected events and data values from the signal generator FB. It compares them with the events and data received from the FB under test.

6) Port test application to other platforms

The presented development approach is fully supported in 4diac IDE [6]. Although service sequences are defined in the standard, they are not fully supported in other tools. Also the model execution framework for evaluating FBs is provided in 4diac IDE. Hence, the test application is generated in 4diac IDE following the notation of the IEC 61499 standard, and is then ported to other platforms. Manual effort may be required for importing FBs developed in one IDE to other vendors [37].

7) Execute Tests in All Relevant RTEs

The generated test application (i.e., the composite FB), is deployed to and executed on different RTEs. The behaviour and output results of the FB are evaluated manually in each RTE.

V. IMPLEMENTATION

Sophisticated tool support can automate a large part of the process, especially for the steps 4 to 6. The systematic approach ensures that FBs can be thoroughly tested for functionality and compatibility across various RTEs. Tool support for specifying service sequences and simulating their results is available in 4diac IDE from previous work [35]. We have extended the tool with a test FB generator, which uses the information provided in service models to create test code. Unlike the prototype presented in [14], the current implementation fully automates the process of creating control code and supports all kinds of test cases that can be modeled in service sequences. The code is available open source on Github.[1]

A. CASE STUDY: PROCESSING STATION

The processing station (Figure 5) is composed of several mechatronic components, including the table, tester, and drill

[1]https://github.com/eclipse-4diac/4diac-ide/tree/release/plugins/org.eclipse.fordiac.ide.fb.interpreter

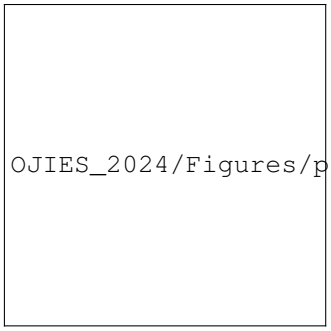


FIGURE 5. Processing station

component. They are considered smart, i.e., each is equipped with its own control devices, implementing the provided operations. The table component undergoes rotation from one fixed position to another. A complete cycle is achieved when the table rotates six times. Whenever a material is positioned in the loading area, the table rotates to align it underneath the tester component. The tester then checks whether the material has been drilled. If necessary, the drill component is triggered to process the material as soon as its sensor detects it.

The processing station system implemented in IEC 61499 is designed to control various mechatronic components through an integrated application. Each of these components is equipped with its own control device, which execute distinct control programs, i.e., the TableControl, TesterControl, and DrillControl programs. The application is implemented following the "Chain of Actions" design pattern [38], which draws inspiration from the "Chain of Responsibility" design pattern. Each of the components is implemented as a FB network that is shown in Figure 7. For the evaluation, three Basic FBs controlling the respective substations will be presented in detail.

1) Table Control: Rotation

The TableControl FB network (cf. Figure 7 *top*) is responsible for rotating the table to position the material appropriately under the tester and drill components. The `WPdeliveryService` FB, a composite FB encapsulating another FB network, manages this rotation through the `TableDriver` FB and a pair of `E_DELAY` FBs. The latter are part of the core FB library defined in the standard. Multiple instances of `WPdeliveryService` are included in the control application.

The `TableRotate` FB encapsulates the core control logic within the network, making it the primary focus for testing the component's features. Its interface and state diagram are shown in Figure 6. The FB controls the rotational movement of a table machine which can rotate to different positions as required by the system's operation through state transitions. The primary functions include starting the rotation, monitoring the rotation process, checking if the table is in the correct position, and handling timeouts. This FB ensures that the table's motion is accurately controlled and

stops when the desired position is reached or if a timeout occurs. As shown in the FB interface, the main events are ROTATE and TIMEOUT_EXCEED. The inPosition input variable signals that the table has reached its target position, while output events such as DRIVE_ON, DRIVE_OFF, and DONE manage the rotation process. The state diagram (i.e., ECC) initially has an active START state, moves to ROTATE_START when the rotation begins, and transitions to ROTATE_CONTINUE if a timeout occurs. Once the table is in position, the FB transitions to DONE, stops the rotation, and then returns to START, ensuring precise control of the table's movement. Two test scenarios for the timeout are included as service sequences in Figure 6. They show that the drive is switched off if the position has been reached. The scenarios ensure that the rotation is stopped even if the signal inPosition is not received correctly.

2) Tester Control: Inspection

The FB network for the Tester component, TesterControl, detects holes in the workpiece to prevent that a workpiece is drilled more than once and to verify that the workpiece has undergone drilling. The main component is the Composite FB Test which is comprised of a standard library FB (E_DELAY) and the TestCtrl Basic FB.

Upon activation, the TestCtrl FB uses sensor data to check whether a hole is present in the workpiece. In this case, it confirms that drilling was complete. The FB manages the sequence of extending, checking, and retracting the probe, and handles timeouts in case the process takes too long. The interface and ECC of the FB are shown in Figure 8. The primary input event, TRIGGER, initiates the detection sequence, with QI qualifying this event and QO reflecting the detection result. The FB has four output events: EXTEND to extend the sensor, RETRACT to retract it, DONE to signal that the process was completed, and TIMEOUT to initiate a timer. The ECC transitions from START to EXTEND upon receiving TRIGGER, and if a timeout occurs, moves to CHECK to determine whether a hole was detected. It then transitions to RETRACT and finally to DONE. A test case that shows the behaviour in the case of a timeout is shown in Figure 8.

3) Drill Control: Processing

The FB network for the drill component, DrillControl, orchestrates the drilling process. The main FBs are DrillingSequence and DrillDriver. The DrillingSequence is a composite FB that uses two FBs, DoubleActingCylinder and SingleActingActuator. The former is responsible for controlling the vertical movement of the drill, while the latter controls the rotation of the drill motor. As the DoubleActingCylinder FB contains the core logic of the drilling process, it is shown in Figure 9. It controls the bidirectional motion of a drill machine, specifically managing its upward and downward movements. This control is essential for operating machinery that uses linear actuators, such as hydraulic or pneumatic cylinders, to extend and retract, corresponding to the downstroke and upstroke of the drill. The FB handles initialization, execution requests, extension (downward movement), and retraction (upward movement), with input conditions determining the cylinder's state transitions and output commands controlling the actuators. Specifically, the cylinder movement is controlled via the events EXTEND and RETRACT. The input variables atHome and atEnd indicate the cylinder's position, while the output variables extend and retract command its motion. The ECC ensures that the cylinder moves correctly based on events and position feedback. The algorithms activate or stop the cylinder's movement, ensuring precise control for upward and downward motion.

FIGURE 6. TableRotate: Interface, state diagram, and selected test cases.

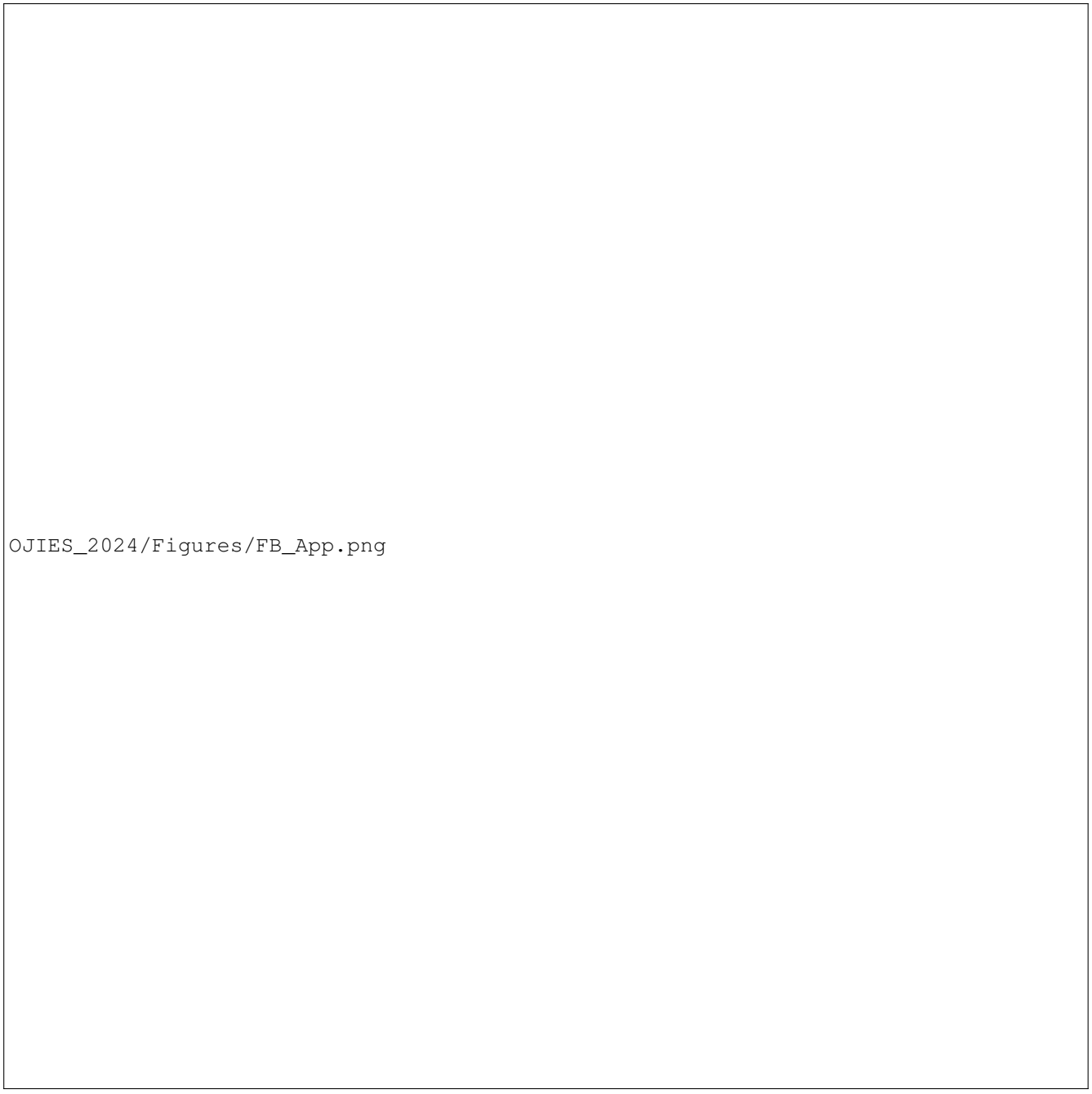


FIGURE 7. Control applications for the three substations including input FBs for processing signals from the physical system, control blocks orchestrating the station, and output FBs for writing information to actuators.

VI. EVALUATION

We tested the main control FBs from the processing station, but also validated the implementation itself. Following our methodology, we imported any FBs from other tools into 4diac IDE. In 4diac IDE, we then generated the test application, which was executed in 4diac FORTE and in EcoRT.

A. EVALUATING THE GENERATION FRAMEWORK IN ECLIPSE 4DIAC

Selected test cases defined as service sequences as shown in Figures 6, 8 and 9. In total, 12 test cases were created which described both the expected interactions with an FB and deviations (such as events from the environment that occur in a reversed order). To additionally validate that the framework can successfully detect failed test cases, we intentionally defined five service sequences with unexpected behaviour. An example for the TableRotate FB is shown in Figure 10.

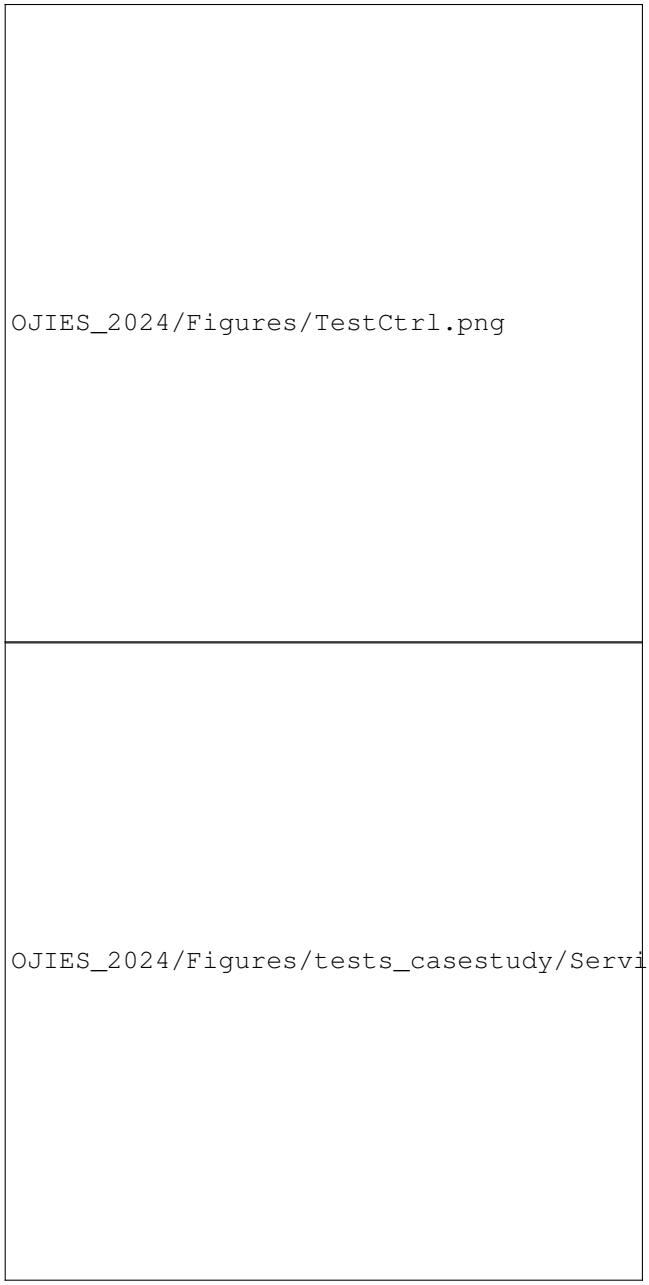


FIGURE 8. TestCtrl: Interface, state diagram and a selected test case.

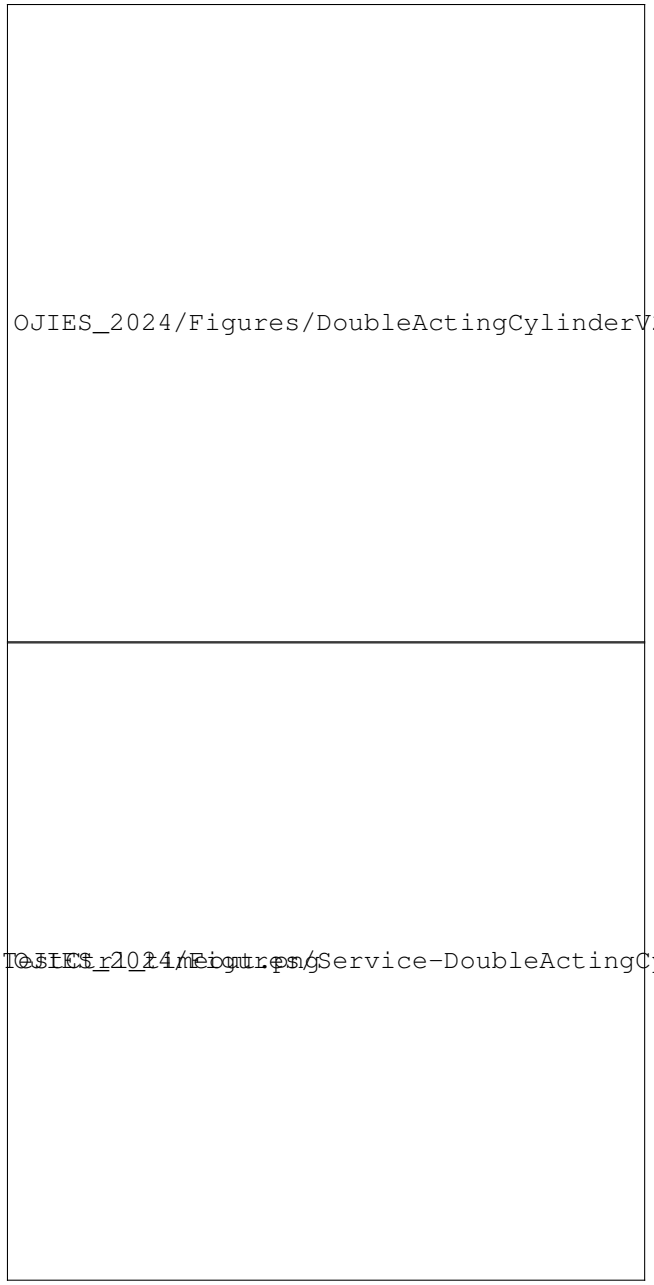


FIGURE 9. Double Acting Cylinder: Interface, state diagram, and selected test cases

These experiments validate that the generated test application can indeed detect deviations between the implementation and the specification.

B. SEMANTIC VARIANTS IN PORTED SOFTWARE

All involved FBs were manually imported in a second tool environment to evaluate the behavior of a ported test application. Unfortunately, the testing of FBs on the EcoStruxure Automation Expert (EAE) revealed that certain test cases were failing when executed on the Tester Control. Modifications to the framework were required to compensate the variations in the implemented execution semantics.

Upon thorough analysis, it was determined that these failures were attributed to a semantic execution issue inherent in the EAE. Unlike specified the IEC 61499 standard, EAE adopts a specific semantic execution model. This divergence in execution semantics implies that when a real-world application is ported from Eclipse 4diac to EAE, it may not function correctly, and in some cases, it may even result in system failures.

EAE's semantic execution model operates such that when two or more identical events are used as a sequence of events to transition through multiple states, a single event trigger results in a direct transition to the final state. Specifically,

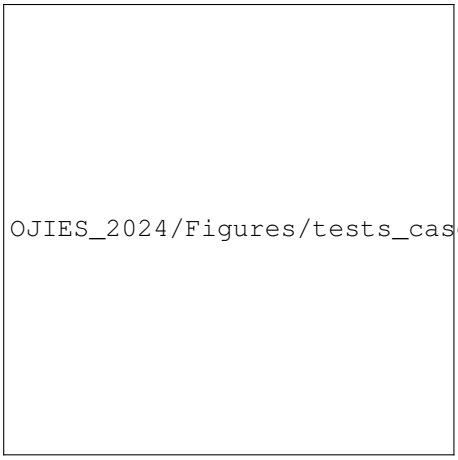


FIGURE 10. Service sequences that result in failed test cases for a correct implementation.

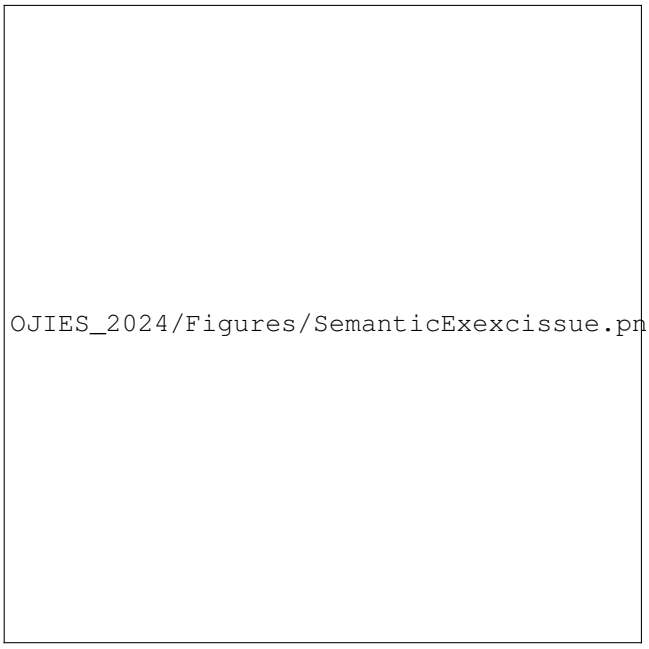


FIGURE 11. Semantic Execution issue

when the event triggers once, the system bypasses intermediate states and directly reaches the final state. To illustrate this issue, consider the ECC shown in Figure 11 where the event `TIMEOUT_EXCEEDED` is expected to facilitate a sequence of state transitions. According to the desired behavior, upon the first occurrence of the `TIMEOUT_EXCEEDED` event, the system should transition to a state labeled `CHECK`. It should remain in this `CHECK` state, awaiting a subsequent `TIMEOUT_EXCEEDED` event to be triggered before progressing to the final state labeled `DONE`. However, due to the semantic execution model employed by EAE, when the `TIMEOUT_EXCEEDED` event is triggered only once, the system bypasses the `CHECK` state entirely and directly transitions to the `DONE` state. This behavior deviates from the intended design, leading to incorrect execution flow.

To address this issue, the test case generation framework was modified to compensate the semantic execution differences. This algorithm modifies the `TIMEOUT_EXCEEDED` event by setting its value to `FALSE`, ensuring that the system remains in the `CHECK` state after the initial trigger. Consequently, the system only moves to the next state when a subsequent `TIMEOUT_EXCEEDED` event is triggered. This modification aligns the execution flow with the expected sequence, preventing premature transitions to the final state.

VII. RESULTS AND DISCUSSION

The semantic differences outlined in the previous section demonstrate the need for a cross-platform testing framework. While the semantic differences could be bridged within our implemented generators, transferring real-world FBs between execution environments can lead to undetected failures. In any portability scenario, this would mean that the distribution of software parts across various vendor's IDEs introduces bugs that are difficult to detect.

It was observed that the control application for our demonstrator contained certain bugs after porting to Eclipse 4diac, which were uncovered during the execution of the test cases. These issues highlight the importance of recognizing the semantic execution differences between EAE and the IEC 61499 standard when porting applications. Addressing such discrepancies is crucial to ensure that applications function as intended within the EAE environment.

The migration of Function Blocks (FBs) from Eclipse 4diac to EcoStruxure Automation Expert (EAE) presents several challenges. Initially, the FBs were imported into 4diac IDE, where tests were defined and test applications were generated. These test applications were executed on the accompanying runtime environment, 4diac FORTE. However, when these test applications were ported to EcoStruxure and executed on the EcoRT runtime environment, portability issues arose. The following outlines the key challenges encountered during this migration process.

One significant challenge involves the direct addition of a composite FB containing the `E_DELAY` FB. The `E_DELAY` FB is a standard function block that is pre-compiled into the vendor's runtime environment. Due to this pre-compiled nature, adding any standard FB directly is not possible. Standard FBs are already available within the vendor's IDE, which necessitates a replacement approach to ensure proper functionality. Consequently, when a composite FB that contains a standard FB is added, it becomes necessary to manually replace the standard FB with an equivalent vendor-provided standard FB within the vendor's IDE.

Another issue encountered pertains to adapters and namespaces. In EAE, an adapter cannot be located unless the correct namespace is specified as `Main`. The definition of the appropriate namespace is essential; otherwise, the adapter will not be displayed. Correcting the namespace can resolve this particular issue. However, even after setting the correct namespace, the use of adapters in composite FBs poses additional problems. These problems require the removal

and subsequent redrawing of connections, indicating that the mere correction of namespaces is insufficient when dealing with composite FBs involving adapters.

Further challenges were identified while porting the test FBs in EAE, particularly in relation to the algorithm section within the `.fbt` file. It was observed that the `start_algorithm` and `end_algorithm` statements were repeated twice. This redundancy results in errors that must be rectified manually by removing the duplicate statements. Additionally, case sensitivity issues arise within the algorithms. For instance, Boolean variable values are written as `"false"` and `"true"` instead of the expected `"FALSE"` and `"TRUE"`. This discrepancy requires manual correction to conform to the syntax standards of the target environment.

Another issue relates to the naming conventions used in algorithms. Algorithm names containing double underscores (__) are not supported. As a result, any algorithm names with double underscores must be modified to eliminate this character sequence. This change is necessary to ensure that the algorithms are compatible with the EAE environment. These migration challenges highlight the need for some manual adjustments to ensure the successful transfer and execution of FBs within different automation environments.

In summary, the proposed framework can reduce the effort of porting libraries because the test execution is automated rather than requiring manual labour.

VIII. LIMITATIONS

We have proposed and implemented a framework for generating cross-platform tests for IEC 61499-based software. We used two tool environments to evaluate the portability and built on previous results of known execution issues. Currently, no industrial-scale FB libraries are publicly available for testing our approach in practice. As described in the paper, service sequences can serve as test cases for IEC 61499 FBs. However, there is no possibility to specify timing information [39]. Our approach mainly targets the event-based behaviour of IEC 61499 FBs, while other approaches can cover timing verification (c.f. [24]).

IX. CONCLUSION AND FUTURE WORK

In conclusion, the proposed testing methodology for IEC 61499 FBs offers systematic and reliable means to verify the correct behaviour of FBs across diverse RTEs. The generation of test FBs from a model-based specification, represented as a service sequence, was accomplished through semi-automated means. Engineers can manually create the test specification (e.g., for test-driven development), or derive it from an existing implementation via the IDE. Our created test FBs are portable across platforms to allow platform-independent testing. This paper presented the overall approach and a first proof-of-concept implementation. We provide an initial set of transformation rules and the corresponding tool support for automating part of the process.

In future work, we aim to support all kinds of FB implementations, provide also the test applications automatically, and evaluate our approach based on a realistic use case to evaluate the scalability and feasibility of the approach in practice. Additionally, developing a runtime comparison tool to analyse and compare the behaviour and performance of FBs across different platforms would enable control engineers to identify and address discrepancies. Finally, integrating the testing approach with further model-based development techniques, such as formal methods or simulation, would provide a holistic approach to system verification and enhance the overall reliability of developed systems.

REFERENCES

[1] G. Lyu and R. W. Brennan, "Towards iec 61499-based distributed intelligent automation: A literature review," IEEE Transactions on Industrial Informatics, vol. 17, DOI 10.1109/TII.2020.3016990, no. 4, pp. 2295–2306, 2021.

[2] M. A. Sehr, M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidig, S. Hoeme, M. Niknami, and E. A. Lee, "Programmable logic controllers in the context of industry 4.0," IEEE Transactions on Industrial Informatics, vol. 17, DOI 10.1109/TII.2020.3007764, no. 5, pp. 3523–3533, 2021.

[3] IEC International Electrotechnical Commission, "IEC 61499-1/ed. 2: Function blocks - part 1: Architecture," 2012.

[4] G. Lyu and R. W. Brennan, "Towards iec 61499-based distributed intelligent automation: A literature review," IEEE Transactions on Industrial Informatics, vol. 17, no. 4, pp. 2295–2306, 2020.

[5] M. Mazzolini, F. A. Cavadini, G. Montalbano, and A. Forni, "Structured approach to the design of automation systems through iec 61499 standard," Procedia Manufacturing, vol. 11, pp. 905–913, 2017.

[6] Eclipse 4diac, "Eclipse 4diac - The Open Source Environment for Distributed Industrial Automation and Control Systems," 2020, Accessed: June 15, 2023. [Online]. Available: www.eclipse.dev/4diac

[7] M. Oberlehner, V. Ashiwal, A. Zoitl, and J. H. Christensen, "Using modules to manage the content of iec 61499 type libraries," in 2022 IEEE 20th International Conference on Industrial Informatics (INDIN), DOI 10.1109/INDIN51773.2022.9976150, pp. 286–292, 2022.

[8] IEC International Electrotechnical Commission, "IEC 61499-2/ed. 2: Function blocks - part 2: Software tool requirements," 2012.

[9] A. Hopsu, U. D. Atmojo, and V. Vyatkin, "On portability of iec 61499 compliant structures and systems," in 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), DOI 10.1109/ISIE.2019.8781290, pp. 1306–1311, 2019.

[10] M. Xavier, T. Liakh, S. Patil, and V. Vyatkin, "Developing a test suite for evaluating IEC 61499 application portability," in 2023 IEEE 32nd Int. Symp. on Industrial Electronics (ISIE), 2023.

[11] B. Wiesmayr, S. Mehlhop, and A. Zoitl, "Close enough? criteria for sufficient simulations of iec 61499 models," in 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE), DOI 10.1109/CASE56687.2023.10260555, pp. 1–7, 2023.

[12] R. Sinha, S. Patil, L. Gomes, and V. Vyatkin, "A survey of static formal methods for building dependable industrial automation systems," IEEE Trans. on Ind. Inf., vol. 15, DOI 10.1109/TII.2019.2908665, no. 7, pp. 3772–3783, 2019.

[13] R. Hametner, I. Hegny, and A. Zoitl, "A unit-test framework for event-driven control components modeled in IEC 61499," in Proc. 2014 IEEE Emerging Technology and Factory Automation (ETFA), DOI 10.1109/ETFA.2014.7005209, 2014.

[14] B. Wiesmayr, M. Xavier, S. Patil, A. Zoitl, and V. Vyatkin, "Generating por table test cases for iec 61499 fbs from interface behaviour specifications," in 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA), DOI 10.1109/ETFA54631.2023.10275633, pp. 1–4, 2023.

[15] P. Jhunjhunwala, S. S. Bitar, K. Zhukovskii, U. D. Atomojo, and V. Vyatkin, "Interoperability in software-defined process automation using the open process automation standard and iec 61499: Adapter connections of iec 61499 and opas enable plug-and-play integration," IEEE Industrial Electronics Magazine, 2024.

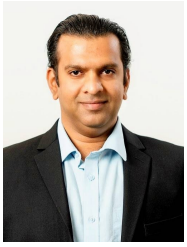
[16] A. Hopsu, U. D. Atmojo, and V. Vyatkin, "On portability of iec 61499 compliant structures and systems," in 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), pp. 1306–1311. IEEE, 2019.

[17] I. Batchkova, G. Popov, H. Karamishev, and G. Stambolov, "Dynamic reconfigurability of control systems using iec 61499 standard," IFAC Proceedings Volumes, vol. 46, no. 8, pp. 256–261, 2013.
[18] G. Lyu and R. W. Brennan, "Multi-agent modelling of cyber-physical systems for iec 61499-based distributed intelligent automation," International Journal of Computer Integrated Manufacturing, pp. 1–27, 2023.
[19] M. Xavier, S. Patil, and V. Vyatkin, "Enhancing traceability in flexible production system: A blockchain-powered approach in iec 61499 multi-agent control system," in 2024 IEEE 33rd International Symposium on Industrial Electronics (ISIE), pp. 1–6. IEEE, 2024.
[20] K. Thramboulidis, "IEC 61499 vs. 61131: A comparison based on misperceptions," Journal of Software Engineering and Applications, pp. 405–415, 2013.
[21] IEC International Electrotechnical Commission, "Iec 61131-3: Programming languages," 2013.
[22] IEC International Electrotechnical Commission, "IEC 61131-10: Plc open xml exchange format," 2019.
[23] M. Marcos, E. Estevez, F. Perez, and E. V. Der Wal, "Xml exchange of control programs," IEEE Industrial Electronics Magazine, vol. 3, DOI 10.1109/MIE.2009.934794, no. 4, pp. 32–35, 2009.
[24] F. Bruns, B. Wiesmayr, and A. Zoitl, "Supporting model-based network specification for time-critical distributed control systems in iec 61499," in 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE), DOI 10.1109/CASE56687.2023.10260604, pp. 1–7, 2023.
[25] B. Lydon, "An open-source control programming platform for universalautomation.org runtime engine," 2024, online. [Online]. Available: https://www.automation.com/en-us/articles/october-2024/open-source-control-programming-platform-universal
[26] G. Cengic and K. Akesson, "On formal analysis of iec 61499 applications, part b: Execution semantics," IEEE Transactions on Industrial Informatics, vol. 6, DOI 10.1109/TII.2010.2040393, no. 2, pp. 145–154, 2010.
[27] C. Pfefferkorn, S. Mehlhop, A. Rauh, and J. Walter, "Towards a methodology for evaluating the execution semantics of iec 61499 runtime environments," in MBMV 2024; 27. Workshop, pp. 175–181, 2024.
[28] M. Xavier, V. Dubinin, S. Patil, and V. Vyatkin, "A framework for the generation of monitor and plant model from event logs using process mining for formal verification of event-driven systems," IEEE Open Journal of the Industrial Electronics Society, 2024.
[29] G. Lilli, M. Xavier, E. Le Priol, V. Perret, T. Liakh, R. Oboe, and V. Vyatkin, "Formal verification of the control software of a radioactive material remote handling system, based on iec 61499," IEEE Open Journal of the Industrial Electronics Society, 2023.
[30] I. Faqrizal, G. Salaün, and Y. Falcone, "Guided Evolution of IEC 61499 Applications," in ETFA 2024 - 29th IEEE International Conference on Emerging Technologies and Factory Automation, pp. 1–8, IEEE. Padova, Italy: IEEEIEEE, Sep. 2024. [Online]. Available: https://inria.hal.science/hal-04680109
[31] T. Hussain and G. Frey, "UML-based development process for iec 61499 with automatic test-case generation," in 2006 IEEE Conference on Emerging Technologies and Factory Automation, DOI 10.1109/ETFA.2006.355407, pp. 1277–1284, 2006.
[32] P. Ovsiannikova, E. Le Priol, V. Perret, P. Jhunjhunwala, M. Xavier, and V. Vyatkin, "Formal verification of observers supervising a cyber-physical system implemented using iec 61499," in 2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE), pp. 1–6. IEEE, 2023.
[33] M. Xavier, S. Patil, V. Dubinin, and V. Vyatkin, "Formal modelling, analysis, and synthesis of modular industrial systems inspired by net condition/event systems," in International Conference on Applications and Theory of Petri Nets and Concurrency, pp. 16–33. Springer, 2023.
[34] G. J. Myers, C. Sandler, and T. Badgett, The art of software testing. John Wiley & Sons, 2011.
[35] B. Wiesmayr, A. Zoitl, A. Garmendia, and M. Wimmer, "A model-based execution framework for interpreting control software," in 26th IEEE Int. Conf. Emerging Technologies and Factory Automation (ETFA), DOI 10.1109/ETFA45728.2021.9613716, 2021.
[36] I. Buzhinsky, V. Ulyantsev, J. Veijalainen, and V. Vyatkin, "Evolutionary approach to coverage testing of IEC 61499 function block applications," in 2015 IEEE 13th Int. Conf. on Industrial Informatics (INDIN), DOI 10.1109/INDIN.2015.7281908, 2015.
[37] C. Pang, S. Patil, C.-W. Yang, V. Vyatkin, and A. Shalyto, "A portability study of iec 61499: Semantics and tools," in 2014 12th IEEE International Conference on Industrial Informatics (INDIN), DOI 10.1109/INDIN.2014.6945553, pp. 440–445, 2014.
[38] S. Patil, D. Drozdov, and V. Vyatkin, "Adapting software design patterns to develop reusable iec 61499 function block applications," in 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), DOI 10.1109/INDIN.2018.8472071, pp. 725–732, 2018.
[39] B. Wiesmayr, F. Roithmayr, and A. Zoitl, "A tool-assisted approach for user-friendly definition of fb constraints," IFAC-PapersOnLine, vol. 56, DOI https://doi.org/10.1016/j.ifacol.2023.10.1531, no. 2, pp. 3666–3672, 2023, 22nd IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896323019390

BIANCA WIESMAYR is a postdoctoral researcher at the Cyber-Physical Systems Lab at the Linz Institute of Technology (LIT) at Johannes Kepler University (JKU) Linz, Austria. She holds a Master's degree in Electronics and Information Technology and a doctoral degree in Engineering Sciences from the same university. Her PhD thesis analyzed the use of behavior modeling in the IEC 61499 development process. Her main research interests include model-driven control software engineering as well as modeling tools and their usability.

ALOIS ZOITL (M'08–SM'22) was born in Wels, Austria in 1977. He received the M.S. and PhD degrees in electrical Engineering from Vienna University of Technology, Austria in 2002 and 2007 respectively. Since 2018 he is professor of Cyber-Physical Systems for Engineering and Production at 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. He is co-author of more than 200 publications (3 books, 6 book chapters, 19 journal articles) and the co-inventor of 4 patents in the mentioned areas. He is a founding member of the open source initiatives Eclipse 4diac, providing a complete IEC 61499 solution, and OpENer. Since 2009 he is an active member of the IEC SC65B/WG15 for the distributed automation standard IEC 61499. He was named convenor of the group in May 2015.

MIDHUN XAVIER received B.Tech in Electronics and Communication Engineering from MG University, Kottayam, India in 2014; the Master of Computer Science from Indian Institute of Information Technology, NIT Trichy Campus, India in 2017. Currently, he is a Ph.D. student at Luleå University of Technology, Luleå, Sweden, with a major in formal verification and modeling of industrial automation systems using IEC 61499 standard. He is also an accomplished software engineer with 3 years of experience in data analytics and web3 development. He has worked with several esteemed organizations such as Uvionics Pvt. Ltd., TCS, and RCKR Software Pvt. Ltd. in India as a Software engineer.



SANDEEP PATIL (S'11, M'19) received a Bachelor's degree in computer science engineering from the CMR Institute of Technology, Bangalore, India, in 2005; a Master of computer science (software engineering) degree from the Illinois Institute of Technology, Chicago, IL, USA, in 2010; the Master of Engineering Studies (computer systems) degree from the University of Auckland, Auckland, New Zealand, in 2011; and a Ph.D. degree in formal verification of cyber-physical systems from the Lulea University of Technology, Lulea, Sweden. His research interests include software engineering principles and methodologies in distributed industrial automation, especially using the IEC 61499 paradigm. He also works with formal verification techniques in the same application field. He is an accomplished software engineering professional with over 16 years of research and development experience in systems and application software, including four years as a Senior Software Engineer at Motorola India Pvt. Ltd., India.

VALERIY VYATKIN (Fellow, IEEE) received Ph.D. degrees in Russia and Japan, in 1992 and 1999, respectively, and the Habilitation degree in Germany, in 2002., He is currently on Joint Appointment as the Chaired Professor with the Luleå University of Technology, Luleå, Sweden, and a Full Professor with Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar at Cambridge University, Cambridge, UK, and had permanent academic appointments with New Zealand, Germany, Japan, and Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures, and multi-agent systems applied in various industry sectors, including smart grid, material handling, building management systems, data centres, and reconfigurable manufacturing. Dr. Vyatkin was a recipient of the Andrew P. Sage Award for the Best IEEE Transactions Paper in 2012. He has been Chair of the IEEE IES Technical Committee on Industrial Informatics since 2016 and Vice President of IES for Technical Activities for the term 2022–2025.