Adaptive Industrial Control Systems via IEC 61499 and Runtime Enforcement

Irman Faqrizal, Gwen Salaün, Yliès Falcone

Adaptive Industrial Control Systems via IEC 61499 and Runtime Enforcement

IRMAN FAQRIZAL, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, France
GWEN SALAÜN, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, France
YLIÈS FALCONE, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, France

This work envisions industrial control systems that can reliably adapt to requirements. We rely on the international standard IEC 61499 to achieve this goal. The standard allows downtimeless system evolution such that an application can be modified at runtime to satisfy the requirements. However, an IEC 61499 application consisting of multiple Function Blocks (FBs) can be modified in many different ways, such as inserting or deleting FBs, creating new FBs with their respective internal behaviours, and adjusting the connections between FBs. These changes require considerable effort and cost, and there is no guarantee to satisfy the requirements. This paper applies runtime enforcement techniques for supporting adaptive IEC 61499 applications. This set of techniques can modify the runtime behaviour of a system according to specific requirements. Our approach begins with specifying the requirements as a state machine-based notation called contract automaton. This automaton is then used to synthesise an enforcer as an FB. Finally, the new FB is integrated into the application to execute according to the requirements. A tool support is developed to automate the approach. Experiments were performed to evaluate the performance of enforcers by measuring the execution time of several applications before and after the integration of enforcers.

CCS Concepts: • **Applied computing** → *Industry and manufacturing*; • **Software and its engineering** → *Formal methods.*

Additional Key Words and Phrases: Adaptive systems, Industrial control systems, IEC 61499, Formal methods, Runtime enforcement

1 Introduction

Manual adaptation of software systems to satisfy specific requirements is both costly and time-consuming, whereas a closed-loop self-adaptive system can take actions automatically based on the observation of the system itself and its environment [48]. The current state of Industrial Control Systems (ICSs) is still relatively far from possessing such characteristics. As surveyed in [63], adaptation technologies used by the industry are mostly trivial, such as auto-scaling and auto-tuning. Stakeholders believe that a fully self-adaptive ICS can be unsafe, and exhaustive human supervision is vital. Moreover, developing and maintaining such a system can be challenging due to the uncertain external aspects of its execution environment [11, 62]. Nevertheless, Industry 4.0 envisions adaptable and resilient systems [58]. An application must be able to reliably adapt to specific requirements with minimal effort.

There exist standards for developing ICSs, such as International Electrotechnical Commission (IEC) 61131-3 [16] and IEC 61499 [15]. IEC 61131-3 defines ladder and function block diagrams, structured text, and sequential function charts, whereas IEC 61499 describes the control logic using interconnected Function Blocks (FBs). In terms of how the system executes, the former adopts a scan-based model where the CPU scans what is to be executed after a certain period to

Authors' Contact Information: Irman Faqrizal, irman.faqrizal@inria.fr, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France; Gwen Salaün, gwen.salaun@inria.fr, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France; Yliès Falcone, ylies.falcone@univ-grenoble-alpes.fr, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France.

Manuscript submitted to ACM

make the system evolve. The latter has an event-driven execution model in which FBs can receive events to trigger their encapsulated functionality.

This work focuses on IEC 61499 because it can efficiently facilitate the course towards adaptive ICSs. As stated in [55], the IEC 61499 standard supports downtimeless system evolution that permits us to change the application (e.g., the addition of FBs) to make it satisfy the requirements without stopping its execution. Furthermore, IEC 61499 has gained more popularity due to its positive characteristics: reusability, reconfigurability, interoperability, and portability [60]. The standard supports distributed design by assigning a single application to multiple control devices. Also, there exist numerous works [12, 17, 29] that propose the integration and migration methods from IEC 61131-3 into IEC 61499 systems. This shows that researchers and stakeholders in this domain encourage the use of this recent standard.

The application of formal methods [66] in supporting the reliability and robustness of ICSs is a continuous research topic [67]. In [6, 19, 30, 32], the authors apply model checking techniques [5] to the aforementioned standards. This method efficiently supports the development of control programs during design time by ensuring their correctness, but it is not ideal for making them adapt to requirements after deployment. The authors in [44–46] propose techniques that take initial and target applications as input to generate dynamic modification sequences. Complementarily, the works in [52, 53, 65] introduce design patterns for supporting the adaptability of IEC 61499 by constructing reusable FBs. These two sets of works can help modify a given initial application into a target application. However, they do not define how to obtain the target application systematically. Hence, the user must provide it as input. Consequently, there is no notion of requirements that should be satisfied after the application has been modified.

IEC 61499 is a promising standard that simplifies the development of ICSs. However, to the best of our knowledge, an approach that can make an IEC 61499 application reliably adapt to requirements has not yet been proposed. This work aims to support the adaptability of systems developed using this standard. Given an application and a set of requirements, our approach results in a new application satisfying the requirements. More precisely, this paper focuses on the following two research questions:

- How can IEC-61499-based ICSs reliably adapt according to requirements?
- How applicable is the solution in terms of performance?

To answer the first question, we rely on runtime enforcement [20, 23], which can make a system adapt to requirements by modifying its execution. Our approach targets functional requirements describing the application behaviour. The requirements are expressed as an input specification. This specification is used to synthesise an *enforcer*. The enforcer is then integrated into the application to modify the runtime execution according to the specification. Specifications can be expressed using temporal logic [43, 70] or Finite State Machine (FSM)-based [36, 49] languages. In this work, we introduce a language called *contract automata*, which is FSM-based. The different types of transition in this language allow the user to specify the modifications of the application executions. A single execution called an action, can be forwarded, discarded, replaced, or buffered. A contract automaton is then used as input for synthesising an enforcer in the form of a basic FB. An enforcer synthesis is a process of constructing all elements of a basic FB, including the interfaces and the execution control chart. The synthesised enforcer is then integrated into the application by appropriately connecting the event and data interfaces. We demonstrate the proposed techniques on an IEC-61499-based conveyor test station [68]. This paper shows that our approach is reliable by ensuring that it conforms to classic runtime enforcement characteristics, such as *soundness* and *transparency*. In addition, we describe how to use model checking to ensure beforehand that the application will not stop progressing (i.e., *deadlock-freedom*) when the enforcers are integrated.

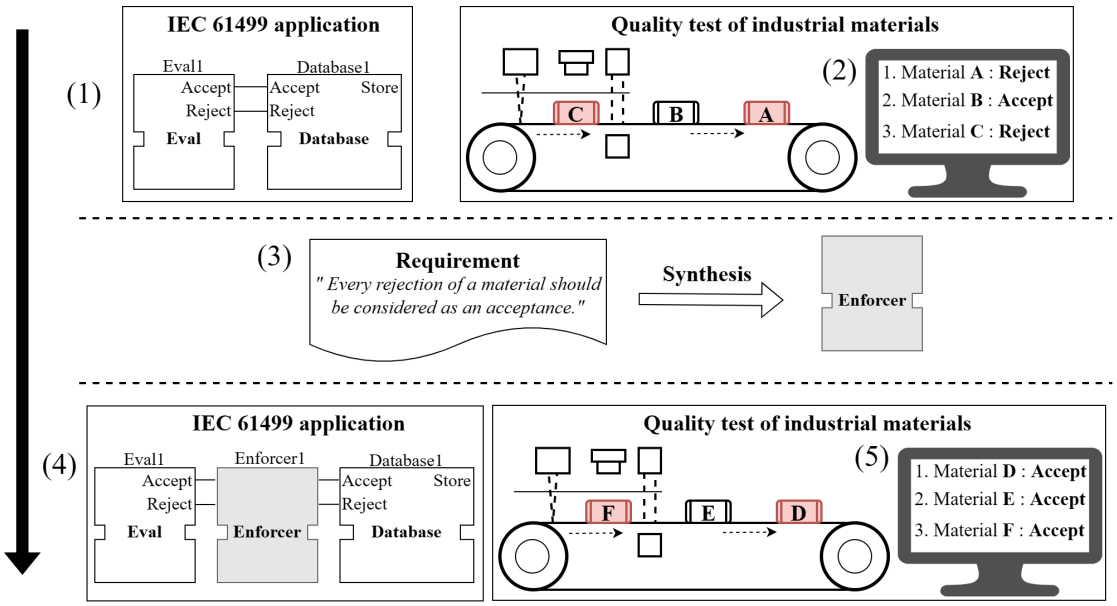


Fig. 1. Runtime enforcement for IEC 61499 in a nutshell

In addition to reliability, we show that our method is applicable in terms of performance. More precisely, the addition of enforcers does not induce significant overheads during the application execution. A toolchain that implements the approach starting from the synthesis of enforcers until the simulation of the application is developed. Experiments were performed to see the differences in application execution time before and after the addition of enforcers. The results show that few enforcers induce overheads. Nevertheless, they are negligible (less than a second).

Figure 1 illustrates the main idea of this work. The industrial system in (2) is a simplified version of the running example introduced later in the paper. Its goal is to evaluate the qualities of materials passing through a conveyor belt and store the results in a database. Defective materials such as A and C are shaded to distinguish these qualities. The evaluation results stored in the database are shown on a screen next to the conveyor. The IEC 61499 application for this design is depicted in (1). It is composed of two FBs. `Eval1` evaluates the quality of each material, whereas `Database1` stores the result in a database.

As illustrated in (2), materials A and C are defective. Hence, they are rejected, and their information is stored in the database accordingly. Suppose there is a situation where the industry's supply chain must receive as many materials as possible. Therefore, quality is not the primary concern, which means every material should be accepted. A conventional practice to take care of this is to change the internal components of either `Eval1` or `Database1` FBs. This method possesses a great risk of introducing bugs to the application, in particular when both FBs are complex composite FBs (i.e., FBs that are composed of networks of FBs). We propose a solution where an enforcer is integrated into the application to change its execution such that the given requirement is satisfied. More precisely, the first step of the approach is to let the user provide a certain requirement. As seen in (3), the user wants to modify every rejection as acceptance. This requirement is then synthesised as an enforcer. Afterwards, in (4), the enforcer is integrated into the application. Every time this enforcer receives an input, it applies the necessary modification to satisfy the requirement. As an outcome in (5), all the materials stored in the database are now accepted regardless of their qualities.

The above example shows that our approach integrates a single FB as an enforcer for each adaptation. In comparison, manual adaptations can be applied in many ways (e.g., by adding or modifying FBs and their connections), which could make the application more complex and error-prone. Furthermore, we propose the notion of composite enforcers to help preserve the readability of applications with multiple enforcers (because they have gone through several adaptations). A composite enforcer is a composite FB consisting of several enforcers. It helps to preserve readability by reducing the number of enforcers at the application level.

In this paper, we introduce runtime enforcement for IEC 61499 to support the adaptation of ICSs to satisfy their requirements. More specifically, the contributions of this paper are summarised as follows.

- A language called contract automata for specifying the requirements of IEC 61499 applications.
- A description of the enforcer synthesis.
- A discussion on the preserved runtime enforcement characteristics, including *soundness*, *transparency*, and *deadlock-freedom*.
- A tool support that implements the synthesis of enforcers.
- A report on the experiments for analysing the impact of enforcers on the application in terms of performance.

The remainder of this paper is organised as follows. Section 2 introduces background notions. Section 3 presents the running example. Section 4 defines a formal model of the IEC 61499 standard. Section 5 describes the runtime enforcement problem. Section 6 explains the runtime enforcement techniques. Section 7 presents the implementation. Section 8 surveys related works. Section 9 concludes.

2 Background

This section discusses the IEC 61499 standard, the runtime enforcement techniques, and requirement classes.

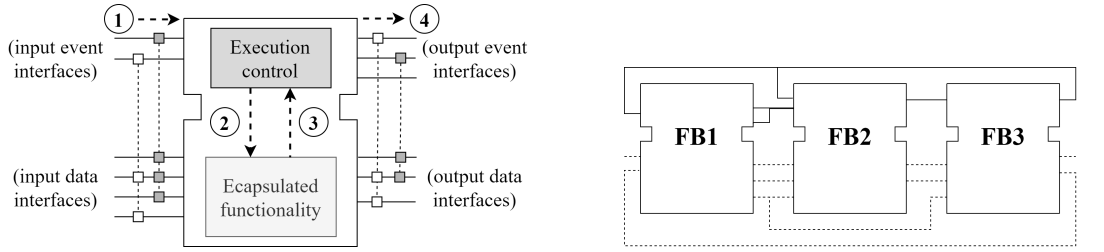
2.1 IEC 61499

IEC 61499 [15] is a standard for designing industrial control systems. In this standard, the behaviour of an application is defined using interconnected function blocks. A Function Block (FB) is connected to other FBs through its input and output event and data interfaces (see Figure 2). Each FB encapsulates an internal behaviour to be executed when one of its input event interfaces receives an event. An FB may also have internal variables.

The execution model of an FB in IEC 61499 standard [68], depicted in Figure 2 (a), is as follows: (1) an event is received, and the values of the associated data interfaces are updated, (2) based on the FB's current state in the execution control, an encapsulated functionality is executed, (3) this execution assigns new values to the output data interfaces, and the execution control determines the output event to be sent, (4) the output event is sent, and the values of the associated output data interfaces are updated. This is the generic execution model. In practice, steps 2 to 4 can be repeated multiple times depending on the FB's type and the execution control. Furthermore, a single activation of the FB can be defined as the period between the beginning of step 1 and the termination of step 4. Once an input event activates the FB, another event cannot enter before the previous activation has finished.

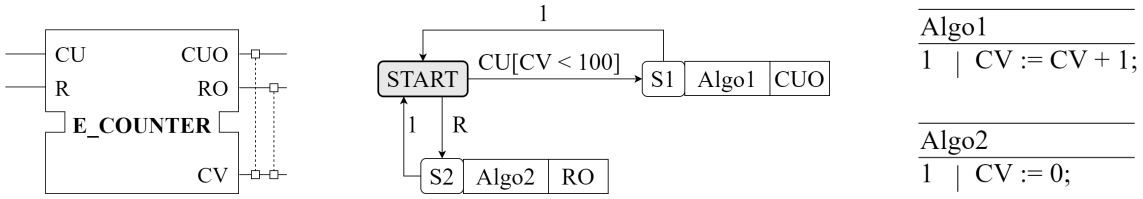
The IEC 61499 application is illustrated with an example in Figure 2 (b). For clarity, connections between event and data interfaces are represented with different styles: solid lines for event connections and dashed lines for data connections. The standard allows both *fan-in* (e.g., `FB1` to `FB2`) and *fan-out* (`FB3` to `FB1` and `FB2`) connections for event interfaces. However, it only allows *fan-out* (e.g., `FB1` to `FB2` and `FB3`) for data interfaces.

There are three types of FBs: basic, service interface, and composite. A basic FB defines its behaviour using a state machine called the Execution Control Chart (ECC). A service interface FB has a behaviour specific to its control device.



(a) FB's execution model (b) IEC 61499 application

Fig. 2. FB and IEC 61499 application



(a) FB and its interfaces (b) ECC (c) Algorithms

Fig. 3. E_COUNTER FB

A composite FB is composed of a network of FBs. In this section, we focus on the description of basic FB since an enforcer is synthesised as an FB with this type. However, our approach can be applied to applications composed of any type of FB.

In a basic FB, the aforementioned execution control is specified using ECC, and encapsulated functionality is expressed as algorithms written in Structured Text [4]. In Figure 3, an example of a basic FB called E_COUNTER [1] with its ECC and algorithms is presented. This FB behaves like a counter. It counts how many times *CU* receives an event and shows the result on *CV*. An example of this FB activation is as follows. When an event arrives on *CU*, the FB is activated. First, it checks the current state of the ECC. By design, the current state of this FB in every activation is always *START*. From this state, if *CV* is less than 100, then *S1* is visited, and *Algo1* is executed to increment *CV* by one. Afterwards, an output event *CUO* is sent. Transition labelled with 1 (i.e., *true*) is automatically traversed to come back to *START*, and then the activation terminates. When *R* receives an event, the transition from *START* to *S2* is traversed without checking the value of *CV*. Then, *Algo2* sets the value of *CV* to 0, and event *RO* is sent. A more detailed explanation of the basic FB's execution model, including the ECC Operation State Machine (OSM), is given in [18].

2.2 Runtime Enforcement

Runtime enforcement [20, 23] is a set of techniques for enforcing certain requirements to the application's behaviour by modifying its runtime execution. This modification is instrumented by an additional component named *enforcer*.

The conceptual view of runtime enforcement is depicted in Figure 4. An enforcer receives as input a trace $\sigma \in \Sigma^*$, a sequence of executions produced by the system. Σ^* represents the set of all possible sequences of executions. The enforcer encapsulates an Enforcement Mechanism (EM), which determines the modification of the trace. The enforcer

[1] This is a simplified version of an FB from the standard library called E_CTU.

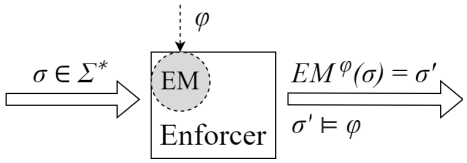


Fig. 4. Conceptual view of runtime enforcement

and its EM are synthesised from requirements specified as φ. EM is defined as a function $EM^{\varphi} : \Sigma^* \rightarrow \Sigma^*$, where for a given input trace σ, it returns an output trace σ' that satisfies the requirements φ, i.e., $EM^{\varphi}(\sigma) = \sigma'$, such that $\sigma' \vDash \varphi$.

Let us illustrate runtime enforcement with an example within the context of IEC 61499. Suppose a trace of the FB `Eval` presented in Figure 1 is defined as the sequence of events triggered from the output event interfaces. For instance, it produces a trace $\sigma = \mathit{Reject}, \mathit{Accept}, \mathit{Reject}$. Requirements can be formulated using LTL (Linear Temporal Logic), as an example $\varphi = \Box \mathit{Accept}$. It means that the application should only generate sequences of *Accept*. Let us assume that EM replaces every *Reject* with *Accept*. Therefore, $EM^{\varphi}(\sigma) = \sigma' = \mathit{Accept}, \mathit{Accept}, \mathit{Accept}$.

The example above is given only to illustrate how runtime enforcement techniques work in practice. The following sections discuss the notions of trace and requirements in more detail.

2.3 Classes of Requirements

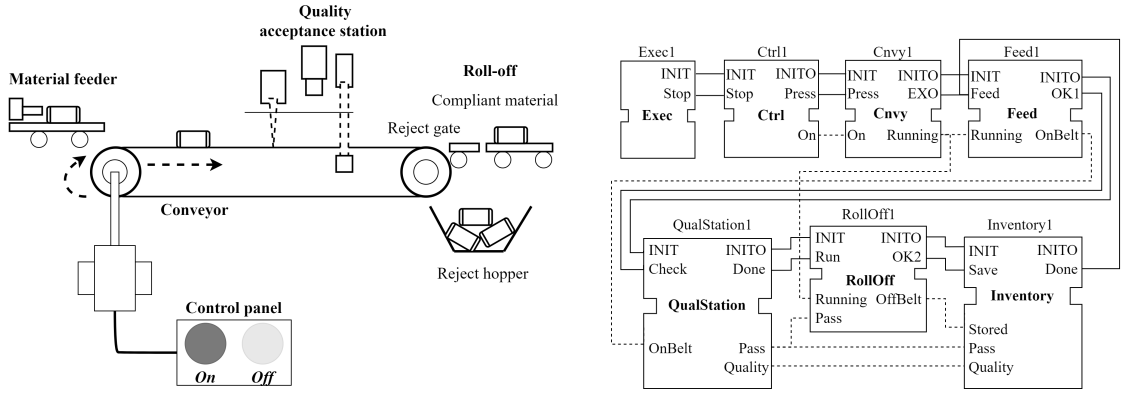
In the software engineering community, requirements are defined as conditions specified by the user to be possessed by the system for achieving an objective [3]. The taxonomy described in [28] classifies requirements into functional requirements, performance requirements, specific quality requirements, and constraints. The last three classes are often categorised as non-functional requirements. A functional requirement is associated with the system's behaviour, for instance, how the system should react when a certain input is given. Performance requirements specify measurable aspects, such as the processing speed of the system. Specific quality requirements define characteristics that the system must have, such as usability and reliability. Constraints are a set of restrictions the system must comply with.

The approach proposed in this paper is aimed at functional requirements, specifically, the ones that involve the system's behaviour at the application level. More precisely, we target requirements that can be satisfied by modifying the sequences of events and data in the IEC 61499 application. For example, let us consider the system presented in Figure 1. We may specify the system to accept a material after two rejections in a row to avoid too many rejected materials. However, we do not consider a requirement that specifies the system to store data in a different database.

3 Running Example

A running example is used in the rest of the paper to illustrate our approach. The example is a conveyor test station, one of the case studies of real-world IEC 61499 applications introduced in [68]. Minor modifications are made to facilitate the explanation of our approach.

Figure 5 (a) presents the physical design of the running example. The system aims to check the qualities of industrial materials passing through a conveyor belt. Firstly, a conveyor is connected to a control panel where the user can either start or stop the conveyor (i.e., by pressing on or off buttons). Then, the component feeder is in charge of transferring materials onto the conveyor. Next, a quality acceptance station evaluates the materials as they pass through. Lastly,



(a) Design (b) Application

Fig. 5. Conveyor test station

depending on the test results, the roll-off mechanism allows the materials to be distributed onto the next industrial process or dropped into a hopper by opening the reject gate.

The IEC 61499 application of the conveyor test station is presented in Figure 5 (b). Each physical component is mapped to a composite FB. `Cnvy1` and `Ctrl1` correspond to the conveyor and its control panel, respectively. `Feed1` represents the material feeder, `QualStation1` deals with the quality acceptance station, and `RollOff1` handles the roll-off mechanism. `Exec1` and `Inventory1` correspond to the initialisation of the application and the database which stores the materials data. In all these FBs, event and data interfaces are associated with the application's functionalities. For instance, the *Running* data interface in `Cnvy1` has a boolean value representing the conveyor belt's state. When the conveyor is running, then *Running* = *true*, otherwise *Running* = *false*. The *Quality* data interface has an integer value representing the quality of the material. It ranges from 0 to 100, and the material is accepted if this value exceeds 50.

The application was designed by its authors following the IEC 61499 hierarchical design patterns [69]. As a result, the application consists of seven composite FBs with at least 35 FBs inside of them in total. A more comprehensive description of the conveyor test station can be found in [68].

4 Formal Model of IEC 61499

The IEC 61499 standard does not define how to represent an application with a formal mathematical model [59]. There exist various works on formalising the standard, such as in [18, 35, 40, 51, 61]. One of the earliest attempts in [61] models IEC 61499 using Petri nets, while the authors in [40] propose a computational model of the standard with Ptolemy II software framework. The work in [51] defines synchronous semantics of the standard where the execution of an FB is divided into *ticks*. Each tick consists of several steps, such as the update of FB's input data interfaces.

A formal model of the standard based on abstract state machines is proposed in [18]. We reuse this model to define the syntactic part of our model because it has the right level of abstraction for our approach. However, the notions of physical time, event queues, and flattening are excluded for the following reasons. The physical time in the application is not considered because we focus on functional requirements, which often do not involve timing constraints (unlike performance requirements). Every activation of an FB is assumed to be completed before receiving other events. Thus, event queues are not considered. Enforcers are connected to FBs that are not part of any sub-application or composite

FB. Therefore, it is not necessary to flatten the application. Furthermore, we limit the type of data values in IEC 61499 to either integer or boolean. This is sufficient in terms of expressiveness without any loss of generality.

We adapt the semantics of IEC 61499 from the work in [35], which defines the notion execution trace. Their notion of execution involves every aspect of the application's evolution, such as the change of ECC's current state. In our perspective, only executions at the application level (i.e., transmissions of events and data) are considered since our runtime enforcement does not modify the behaviours encapsulated in the FBs.

In the remainder of this section, the syntactic elements of an IEC 61499 application are described; they consist of each individual FB and the application itself. Then, we explain the execution semantics of an application based on the set of execution traces that the application can produce. The following notations are used in the rest of this paper:

- $\mathbb{Z}$ is the set of integers.
- $\mathbb{B} = \{true, false\}$ is the set of boolean values.
- $\mathbb{ZB} = \mathbb{Z} \cup \mathbb{B}$ is the union of integers and boolean values.
- $\mathbb{ID}$ is the set of identifiers.
- $\mathbb{V}$ is the set of variables, and each $v \in \mathbb{V}$ is a couple (id, zb), where $id \in \mathbb{ID}$ and $zb \in \mathbb{ZB}$ are, respectively, the identifier and the value of the variable.
- $\mathbb{G}$ represents the set of bags. A *bag* can be used to store any type of element, allowing multiple occurrences. For instance, $\mathbb{G}_\mathbb{Z}$ is the set of bags over integers; each bag $b_\mathbb{Z} \in \mathbb{G}_\mathbb{Z}$ can be used to store integers (e.g., $b_\mathbb{Z} = \{1, 1, 2, 3\}$),
- ϵ represents an empty string.

Syntax. Let us define the syntactic components of an FB as the building block of an IEC 61499 application. Since our enforcer is in the form of a basic FB, it is necessary to include the components of this particular FB, that is, the ECC, algorithms, and internal variables.

Definition 4.1. (Function Block) An FB is defined as a tuple (itf, Ag, DV, ecc), where:

- $itf = (EI, EO, DI, DO, WI, WO)$ consists of input and output event and data interfaces $EI, EO \subseteq \mathbb{ID}$, $DI, DO \subseteq \mathbb{V}$, and relations between them $WI \subseteq EI \times DI$, $WO \subseteq EO \times DO$,
- Ag is a set of algorithms, each $ag \in Ag$ is a sequence of assignments generated by the grammar $ag ::= ag^1.ag^2 \mid x^1 := x^2; \mid x := y; \mid x^1 := x^2 + z; \mid x^1 := x^2 - z;$, where $x \in \mathbb{ID}$, $y \in \mathbb{ZB}$, and $z \in \mathbb{Z}$,
- $DV \subseteq \mathbb{V}$ is a set of internal variables,
- $ecc = (S_{ec}, s^0_{ec}, G_{ec}, T_{ec})$ is the ECC, where:
  - S_{ec} is a set of states, and each $s_{ec} \in S_{ec}$ consists of state actions $s_{ec} = q^1_{ec}, q^2_{ec}, ..., q^n_{ec}$, each action $q_{ec} = (ag, eo)$ consists of an algorithm $ag \in Ag$ and output event interface $eo \in EO$,
  - $s^0_{ec} \in S_{ec}$ is the initial state,
  - G_{ec} is a set of guards; each guard $g_{ec} \in G_{ec}$ is a function $g_{ec} : ei \times DI \times DO \times DV \rightarrow \mathbb{B}$, where $ei \in EI$,
  - $T_{ec} \subseteq S_{ec} \times G_{ec} \times S_{ec}$ is a set of transitions.

The components of `E_COUNTER` FB (introduced in Section 2.1, Figure 3) are shown in Table 1 to illustrate the above definition. There is no internal variable in this FB. Therefore, DV is empty.

When an FB is not a basic FB, then Ag, DV, and ecc are all empty. The algorithms in Ag are written in Structured Text, a language dedicated to industrial automation. We do not focus on the details of this language. Its syntax and semantics are available in [4]. In our work, each algorithm is a sequence of assignments, additions, and subtractions. This is sufficient to construct the algorithms in the enforcers' ECCs to assign data values and increment/decrement

Table 1. Components of E_COUNTER FB

itf	$= (\{CU, R\}, \{CUO, RO\}, \emptyset, \{CV\}, \emptyset, \{(CUO, CV), (RO, CV)\})$
Ag	$= \{ag_1 = \text{“}CV := CV + 1\text{”}, ag_2 = \text{“}CV := 0\text{”}\}$
DV	$= \emptyset$
ecc	$= (S_{ec}, s^0_{ec}, G_{ec}, T_{ec})$
S_{ec}	$= \{START = \emptyset, S1 = \{(ag_1, CUO)\}, S2 = \{(ag_2, RO)\}\}$
s^0_{ec}	$= START$
G_{ec}	$= \{g^1_{ec} = CU[CV < 100], g^2_{ec} = R, g^3_{ec} = 1\}$
T_{ec}	$= \{(START, g^1_{ec}, S1), (S1, g^3_{ec}, START), (START, g^2_{ec}, S2), (S2, g^3_{ec}, START)\}$

buffers. The internal variables in DV can be manipulated in the algorithms and can also be used as guards in the ECC transitions. However, every $dv \in DV$ is inaccessible to other FBs, unlike the ones in DI and DO. Furthermore, dot operators are used to retrieve elements of a tuple. For instance, $fb.itf$ retrieves the interfaces of fb.

We can now define the components of an IEC 61499 application, which consists of FBs and connections between their interfaces.

Definition 4.2. (IEC 61499) An IEC 61499 application is defined as $iec = (FB, EC, DC)$, where $FB = \{fb_1, fb_2, ...\}$ is a set of FBs, $EC \subseteq EO \times EI$ is a set of connections between event interfaces, and $DC \subseteq DO \times DI$ is a set of connections between data interfaces.

Semantics. The runtime behaviour of IEC 61499 is formalised using the notions of action and trace. An action is a pair of an event and data sent by the output interfaces of an FB to the input interfaces of some FBs according to EC and DC. An instance of an event is identified using the name of the output event interface from which this event has been sent, and the data is derived from the associated output data interfaces. A sequence of actions is called a trace.

Definition 4.3. (Action) An action is defined as a pair (e, D), where $e \in EO$ is an event, and $D \subseteq 2^{DO}$ is a set of data. The set of all possible actions that an application can execute is denoted as Σ.

Definition 4.4. (Trace) A trace σ of size n is a sequence of actions $a_1, a_2, ..., a_n \in \Sigma$. The set of all possible traces that an application can execute is denoted as Σ^*.

To illustrate action and trace, let us consider that the running example in Figure 5 (b) executes the following sequence:

(1) In `Ctrl1`, *Press* sends an event and *On* becomes *true* because the user has pressed the on button.
(2) The conveyor is switched on. Therefore, `Cnvy1` sends an event from *EXO* and *Running* = *true*.
(3) The feeder feeds a material onto the conveyor. As a result, `Feed1` sends an event from *OK1* and *OnBelt* = *true*.
(4) The quality station determines that the material's quality is 90. Consequently, an event is sent from *Done* by `Quality1` with *Pass* = *true* and *Quality* = *90*.
(5) The roll-off mechanism puts the material into the hopper. Thus, `RollOff1` triggers an event from *OK2* and *OffBelt* = *true*.
(6) In `Ctrl1`, *Press* sends an event and *On* = *false* since a user pressed the off button on the control panel.
(7) The conveyor stops running. Hence, from `Cnvy1`, *EXO* sends an event with *Running* = *false*.

This sequence of executions can be represented as the following trace:

$$\sigma = (Press, \{(On, true)\}), (EXO, \{(Running, true)\}), (OK1, \{(OnBelt, true)\}),$$
$$(Done, \{(Pass, true), (Quality, 90)\}), (OK2, \{(OffBelt, true)\}), (Press, \{(On, false)\}),$$
$$(EXO, \{(Running, false)\})$$

The set of all traces Σ^* defines the possible runtime behaviour of an application. Function $Run : IEC \rightarrow 2^{\Sigma^*}$ is defined to represent this behaviour, where IEC is a set of IEC 61499 applications. Let us consider our conveyor test station example as an application $iec = (FB, EC, DC)$. Therefore, the above trace $\sigma \in Run(iec)$.

5 Runtime Enforcement Problem for IEC 61499

The goal of our runtime enforcement approach from the structural point of view is to transform an IEC 61499 application $iec = (FB, EC, DC)$ into $iec' = (FB', EC', DC')$. The new set of FBs is $FB' = FB \cup \{fb_e\}$, where $fb_e = (itf_e, Ag_e, DV_e, ecc_e)$ is a basic FB that plays the role of the enforcer according to given requirements. EC' and DC' are the new sets of connections after fb_e is integrated into the application.

From the runtime perspective, our objective is to ensure that the execution of the application satisfies the requirements. More precisely, let us consider the input requirements as φ, and Σ^* is the set of all possible traces that can be generated by iec. The new application iec' should be able to produce a set of traces $Run(iec') = A^* \subseteq \Sigma^*$, such that $\forall \sigma \in A^* : \sigma \vDash \varphi$. It means that the enforcer fb_e, along with all its components $(itf_e, Ag_e, DV_e, ecc_e)$, must be synthesised using a certain mechanism such that it can behave as a function $EM^{\varphi} : \Sigma^* \rightarrow \Sigma^*$. This function modifies every sequence of actions $\sigma = a_1, a_2, ..., a_n \in \Sigma^*$ that would violate the requirements, i.e., $\sigma \nvDash \varphi$, into $\sigma' = a'_1, a'_2, ..., a'_m \in \Sigma^*$, such that $\sigma' \vDash \varphi$. In our case, an action is a pair of an event and data $(e, D) \in \Sigma$. Therefore, the enforcer may modify the event or data.

With respect to the perspectives above, we focus on solving the following problems.

- How can one express the requirements φ for specifying the behaviour of an IEC 61499 application iec?
- How to synthesise an enforcer $fb_e = (itf_e, Ag_e, DV_e, ecc_e)$ according to the requirements φ?
- How to integrate an enforcer fb_e into the application $iec = (FB, EC, DC)$ in order to obtain a new application $iec' = (FB \cup \{fb_e\}, EC', DC')$?

6 Runtime Enforcement Techniques

This section describes the runtime enforcement techniques for IEC 61499 applications. Section 6.1 introduces the input specification called contract automata. Section 6.2 explains the enforcer's synthesis from a given contract automaton. Section 6.3 shows how enforcers are integrated into the application. Section 6.5 discusses the preserved runtime enforcement characteristics.

6.1 Contract Automata

This paper proposes a specification language called contract automata to express the requirements of IEC 61499 applications. More precisely, users can specify a contract automaton representing the new requirements corresponding to situations in which the application must behave differently. The term contract means that the application will behave according to the automaton, whereas automata is a common terminology in runtime enforcement when defining specification languages. For instance, two notable languages are *Security Automata* [49] and *Edit Automata* [36]. Contract automata is an extended version of these two languages where transitions are typed to add expressiveness. The notions

Fig. 6. Transition types in contract automata

of *accepting* and *non-accepting* states are excluded because there is no need to specify the type of each state. Contract automata can be specified to buffer the application executions (i.e., actions). Users may also specify to discard or replace specific executions when necessary. Overall, in this work, contract automata is proposed as an expressive and intuitive runtime enforcement specification language for IEC 61499 applications.

In this section, the syntax of contract automata is first introduced, followed by the description of its semantics. In addition, two examples of contract automata are presented for illustration purposes.

6.1.1 Syntax. A contract automaton consists of states and transitions. The transitions are typed to specify unique modifications to the set of actions Σ that the application can execute. In other words, when an action is executed, the type of transition corresponding to that action determines the modification to be applied. In Figure 6, these types are represented with different styles, and their descriptions are as follows:

- *forward* indicates no modification,
- *discard* indicates that the action is discarded,
- *replace* indicates that the action is replaced with a different one,
- *buffer* indicates that the action is buffered.

Besides a type, a transition also has a guard. The set of all guards present in the automaton is denoted as G, and the set of all actions specified by G is defined as $A \subseteq \Sigma$, where Σ is the set of all actions in the application. A guard $g \in G$ is a function $g : \Sigma \rightarrow \mathbb{B} \cup \{i\}$. The symbol i is returned to denote that the input action $a \notin A$.

To illustrate guards, let us consider actions $a = (Press, \{(On, false)\})$, $a' = (OK1, \{(OnBelt, true)\})$, and the set of guards $G = \{g = Press[On = false], g' = EXO[Running = false]\}$. Given this set of guards, the set of specified actions is $A = \{(Press, \{(On, true)\}), (Press, \{(On, false)\}),$
$(EXO, \{(Running, true)\}), (EXO, \{(Running, false)\})\}$. In this example, $g(a) = true$ and $g'(a) = false$ because $a \in A$ and action a satisfies guard g, but it does not satisfy g'. On the other hand, $g(a') = i$ and $g'(a') = i$ because $a' \notin A$.

Let us define a contract automaton consisting of states and transitions labelled with types and guards.

Definition 6.1. (Contract Automaton) Given a set of actions Σ, a contract automaton is a tuple (S, s^0, A, Γ, T), where:

- S is a (finite) set of states, and $s^0 \in S$ is the initial state,
- $A \subseteq \Sigma$ is the set of specified actions,
- $\Gamma = \{forward, discard, replace, buffer\}$ is the set of transition types,
- T is a set of transitions and each transition $t = (s, g, \gamma, a', s') \in T$, where:
  - $s, s' \in S$ are source and target states,
  - g is a guard encoded as a function $g : \Sigma \rightarrow \mathbb{B} \cup \{i\}$, the set of all guards is denoted as G,
  - $\gamma \in \Gamma$ is the transition type,
  - $a' \in A$ is a replacement action.

The replacement action a' is necessary for transitions typed as *replace* because each transition with this type represents the replacement of an initial action $a \in A$, satisfying the guard g, with a different action $a' \in A$. Transitions typed as *forward*, *discard*, or *buffer* do not indicate replacements of actions; therefore, a' is empty in these transitions.

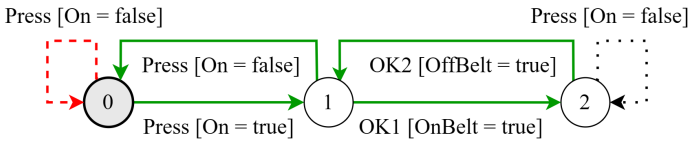


Fig. 7. Automaton A with *discard* and *buffer* transitions

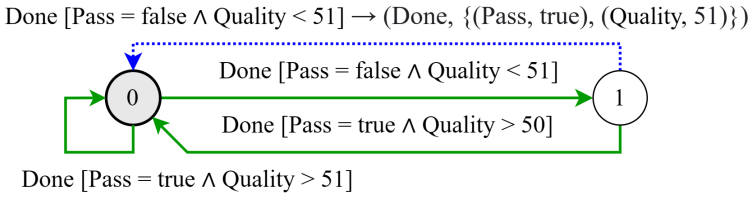


Fig. 8. Automaton B with a *replace* transition

Let us discuss two examples called Automaton A and Automaton B in Figures 7 and 8, respectively, to illustrate the syntax of contract automata. Given the running example presented in Section 3, suppose the following functional requirements are desired:

(i) Pressing the on button when the conveyor is running or off when idle should not impact the application to prevent unexpected behaviour.
(ii) If the button off is pressed when there is still material on the conveyor, then the conveyor will only stop running when that material has passed through the roll-off mechanism to ensure that the conveyor is unoccupied when it stops.
(iii) To avoid too many rejected materials, there should not be two rejections in a row. The second rejection is considered an acceptance instead, and the material's quality is set to 51.

Automaton A aims to satisfy (i) and (ii). For specifying requirement (i), from the initial state, we apply a correction by discarding the action satisfying the guard $Press[On = false]$ to suppress the impact of a user pressing the off button when the conveyor is idle. Transitions typed as *discard* can be implicit when the source and target states are the same (i.e., self-loop). For instance, in state 1, it is not mandatory to specify a *discard* transition guarded with $Press[On = true]$. This discard mechanism is explained further in the semantics part of this section. To satisfy (ii), we specify the transition in state 2 to buffer an action satisfying the guard $Press[On = false]$. This is because in this state, action $(OK1, \{(OnBelt, true)\})$ has been executed (i.e., the material has been fed onto the conveyor) and action $(OK2, \{(OffBelt, true)\})$ has not been executed yet (i.e., the material has not passed through the roll-off mechanism).

Automaton B specifies requirement (iii). In state 1, when the action satisfying the guard $Done[Pass = false]$ (i.e., the quality station rejects the material) is executed for the second time, we use a replace transition to trigger a different action that is $(Done, \{(Pass, true), (Quality, 51)\})$ (i.e., the material is accepted and its quality is set to 51).

6.1.2 Semantics. To define the semantics of contract automata, let us first discuss the notion of buffer. We consider the set of bags over the set of actions Σ as $\mathbb{G}_\Sigma$. A buffer $buf \in \mathbb{G}_\Sigma$ is a bag to store buffered actions. The following functions are used to represent the operations on this buffer. Functions $add : \mathbb{G}_\Sigma \times \Sigma \to \mathbb{G}_\Sigma$ and $remove : \mathbb{G}_\Sigma \times \Sigma \to \mathbb{G}_\Sigma$ are respectively used for adding and removing actions from the buffer. The set of configurations in a contract automaton

Table 2. Transition rules of contract automata

$$\frac{\forall s \xrightarrow{(g,\gamma,a')} s' \in T \quad g(a) = i}{(s, buf) \xrightarrow{a/a} (s, buf)} (f_0) \qquad \frac{\exists s \xrightarrow{(g,forward,\epsilon)} s' \in T \quad g(a) = true}{(s, buf) \xrightarrow{a/a} (s', buf)} (f_1)$$

$$\frac{\forall s \xrightarrow{(g,\gamma,a')} s' \in T \quad g(a) = false}{(s, buf) \xrightarrow{a/\epsilon} (s, buf)} (d_0) \qquad \frac{\exists s \xrightarrow{(g,discard,\epsilon)} s' \in T \quad g(a) = true}{(s, buf) \xrightarrow{a/\epsilon} (s', buf)} (d_1)$$

$$\frac{\exists s \xrightarrow{(g,replace,a')} s' \in T \quad g(a) = true}{(s, buf) \xrightarrow{a/a'} (s', buf)} (r_1) \qquad \frac{\exists s \xrightarrow{(g,buffer,\epsilon)} s' \in T \quad g(a) = true}{(s, buf) \xrightarrow{a/\epsilon} (s', add(a, buf))} (b_1)$$

$$\frac{Find(s, buf) = a_1, a_2, ..., a_n = \beta \quad Reach(s, \beta) = s'}{(s, buf) \xrightarrow{\epsilon/a_1,a_2,...,a_n} (s', remove(\beta, buf))} (f_2)$$

is defined as $Conf = S \times \mathbb{G}_{\Sigma}$, where S is the set of states. A configuration, which is a tuple (s, buf), may change when an action is executed, and it satisfies the guard of the corresponding transition. We define functions related to the release of actions from a buffer. Function $Find : S \times \mathbb{G}_{\Sigma} \rightarrow \Sigma^*$ finds a sequence of actions in the buffer that subsequently satisfies transition guards typed as *forward* from a given starting state. Function $Reach : S \times \Sigma^* \rightarrow S$ returns a reachable state from a sequence of actions that subsequently satisfies transition guards typed as *forward* from a given starting state.

Table 2 presents the transition rules of contract automata. Each rule defines the behaviour of the application every time an action is executed. We use examples of contract automata in Figures 7 and 8 (i.e., Automaton A and Automaton B) to illustrate these rules in the following explanations:

- f_0 states that if every transition guard returns i when evaluating the executed action a, this action is not modified, and the configuration does not change. This is the case when an action $(INIT, \emptyset)$ is executed from any state in Automaton A, this action is forwarded, and the current state does not move.
- f_1 states that when the executed action satisfies a transition guard typed as *forward*, this action is not modified, and the current state moves to the transition target state. As an example, when an action $(Press, \{(On, true)\})$ is executed in Automaton A state 0, this action is forwarded, and the current state moves to state 1.
- d_0 states that when the executed action does not satisfy any of the transition guards, the action is discarded, and the current state remains the same. For example, when an action $(Press, \{(On, true)\})$ is executed in state 1 of Automaton A, this execution is discarded, and the current state remains in state 1.
- d_1 states that when the executed action satisfies a transition guard typed as *discard*, this action is discarded, and the current state transitions to the target state. For instance, when action $(Press, \{(On, false)\})$ is executed by the application in state 0 of Automaton A, this action is discarded, and the current state transitions to the same state since the transition is a self-loop.
- r_1 states that when the executed action satisfies a transition guard typed as *replace*, this action is replaced with a replacement action and the transition is traversed. As an example, when an action $(Done, \{(Pass, false), (Quality, 50)\})$

is executed in state 1 of Automaton B, this action is replaced with $(Done, \{(Pass, true), (Quality, 51)\})$ and the current state moves to state 0.

- b_1 states that when the executed action satisfies a transition guard typed as *buffer*, this action is added into a buffer, and the current state moves to the target state. For instance, when action $(Press, \{(On, false)\})$ is executed in state 2 of Automaton A, this action is buffered, and the current state transitions to the same state since the transition is a self-loop.
- f_2 states that a sequence of actions is released from the buffer when possible. More precisely, when an action in the buffer satisfies a transition guard typed as *forward* from the current state, this action and the next subsequent actions that also satisfy the corresponding transition guards typed as *forward*, are removed from the buffer. Then, this sequence of actions is executed sequentially, and the current state moves to a state where no more action in the buffer can satisfy a transition guard typed as *forward*. For example, in state 1 an action $(Press, \{(On, false)\})$ is available in the buffer. This action satisfies the guard $Press[On = false]$ in transition 1 to 0. Therefore, the action is removed from the buffer and executed. Moreover, the current state moves from 1 to 0.

6.2 Synthesis

Enforcer synthesis transforms a contract automaton in Definition 6.1 into a basic FB in Definition 4.1. An enforcer is denoted as $fb_e = (itf_e, Ag_e, DV_e, ecc_e)$. Synthesis is divided into two main parts: interfaces itf_e and ECC ecc_e. The internal variables DV_e and algorithms Ag_e are included in the synthesis of ECC.

6.2.1 Interfaces. The interfaces of an enforcer are built by analysing the set of specified actions in the contract automaton. A pair of input and output interfaces is created for every event and data present in an action.

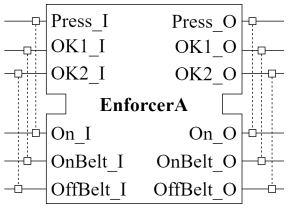
Definition 6.2. (Enforcer Interfaces) The enforcer interfaces $itf_e = (EI, EO, DI, DO, WI, WO)$ are derived from the set of specified actions A in the contract automaton $c = (S, s^0, A, \Gamma, T)$ such that:

- $EI = \{a.e + \text{“\_}I\text{”} \mid a \in A\}$,
- $EO = \{a.e + \text{“\_}O\text{”} \mid a \in A\}$,
- $DI = \{a.D + \text{“\_}I\text{”} \mid a \in A\}$,
- $DO = \{a.D + \text{“\_}O\text{”} \mid a \in A\}$,
- $WI = \{(a.e + \text{“\_}I\text{”},\ a.D + \text{“\_}I\text{”}) \mid a \in A\}$,
- $WO = \{(a.e + \text{“\_}O\text{”},\ a.D + \text{“\_}O\text{”}) \mid a \in A\}$

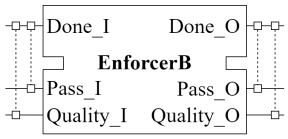
In the definition above, suffixes “_I” and “_O” are appended to, respectively, input and output interfaces. Moreover, a pair of event and data interfaces are added into *WI* and *WO* when they are synthesised from the same action.

Example. Figure 9 presents the synthesised enforcers' interfaces. `EnforcerA` and `EnforcerB` are the FBs synthesised from Automaton A and Automaton B, respectively. As defined above, the interfaces consist of input and output interfaces interpreted from the existing actions. For instance, in `EnforcerA`, *Press_I*, *Press_O*, *On_I*, and *On_O* interfaces are synthesised since action $(Press, \{(On, true)\})$ is specified in Automaton A.

6.2.2 ECC. The enforcer's ECC is constructed using a set of translation patterns to transform each transition in the contract automaton into transitions and states of ECC. However, it is necessary beforehand to identify the release of actions characterised by rule *f2* in Table 2 by tagging the corresponding transitions in the contract automaton. These tags help to determine which transitions in the contract automaton should be translated into ECC's transitions and states that can release the buffered actions.



(a) Interfaces of EnforcerA



(b) Interfaces of EnforcerB

Fig. 9. Synthesised enforcers' interfaces from Automaton A and Automaton B

Fig. 10. Untagged and tagged transitions

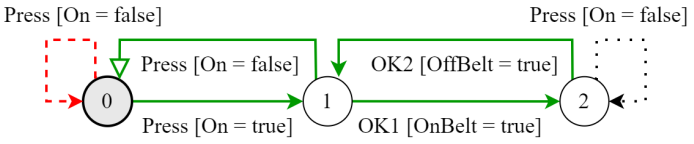


Fig. 11. Tagged Automaton A

The set of tags is defined as $\Lambda = \{u, release\}$, where u signifies an untagged transition while *release* corresponds to transitions that can release the buffered actions. As illustrated in Figure 10, a transition tagged as *release* is represented using a different endpoint style.

Definition 6.3. (Tagged Contract Automaton) Given a contract automaton $c = (S, s^0, A, \Gamma, T)$ and the set of tags $\Lambda = \{u, release\}$, a tagged contract automaton is $c_\Lambda = (S, s^0, A, \Gamma, \Lambda, T_\Lambda)$, where each transition $t \in T_\Lambda$ is extended as $t = (s, g, \gamma, a', \lambda, s'_a)$, where $\lambda \in \Lambda$. Let $Tag : T \to \Lambda$ be the function to tag every transition $t \in T$ in the automaton and $T' \subseteq T$ the set of all transitions typed as *buffer*:

$$Tag(t) \begin{cases} release & \text{if } (t.\gamma = forward) \wedge (\exists t' \in T' \ : \ t.g = t'.g) \\ u & \text{otherwise} \end{cases}$$

Tagged Automaton A is presented in Figure 11. The transition from states 1 to 0 is tagged as *release* because it is typed as *forward* and guarded with the same guard as the transition typed as *buffer* (the self-loop transition in state 2).

A contract automaton that has been tagged can be translated into an ECC. To represent the transformations of the contract automaton's elements into elements of ECC, we define a function $Trans : (G \cup \Sigma) \times \Theta \to G_{ec} \cup Ag \cup EO$, where $\Theta = \{guard, guardRelease, algoForward, algoReplace,$
$algoBuffer, algoRelease, output\}$. Each $\theta \in \Theta$ represents a transformation task from a contract automaton element into an ECC element. For instance, *guard* transforms a contract automaton's guard into an ECC's guard, *algoReplace* transforms a replacement action into an algorithm to replace data, and *algoBuffer* transforms a contract automaton's guard into an algorithm to buffer an action.

To illustrate these functions, let us consider a guard $g = Done[Pass = true \wedge Quality > 50]$ and an action $a = (Done, \{(Pass, false), (Quality, 40)\})$. Hence, the following results are obtained:

- $Trans(g, guard) = Done\_I[Pass\_I = true \wedge Quality\_I > 50]$,
- $Trans(a, algoReplace) =$ "$Pass\_O := false;\ Quality\_O := 40;$", and
- $Trans(g, algoBuffer) =$ "$Buf\_Pass[Buf\_Done] := Pass\_I;$
  $Buf\_Quality[Buf\_Done] := Quality\_I;\ Buf\_Done := Buf\_Done + 1;$"

Translation patterns in Table 3 show how a contract automaton can be translated into an ECC that implements the rules in Table 2. The description of these patterns is as follows:

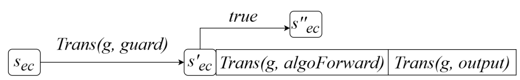
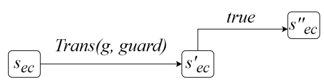
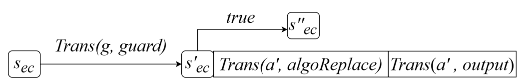
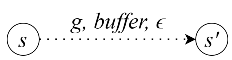
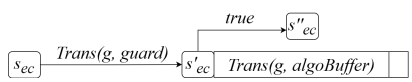
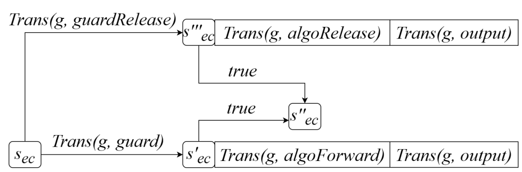
- Pattern (1) is associated with rule f_1. A *forward* transition in a contract automaton is translated to an ECC transition, leading to a state that executes the same action received by the enforcer.
- Pattern (2) is associated with rule d_1. A *discard* transition in a contract automaton is translated to an ECC transition, leading to a state that does not execute any action.
- Pattern (3) is associated with rule r_1. A *replace* transition in a contract automaton is translated to an ECC transition, leading to a state that executes the replacement action.
- Pattern (4) is associated with rule b_1. A *buffer* transition in a contract automaton is translated to an ECC transition, leading to a state that stores the received action in a buffer.
- Pattern (5) is associated with rule f_2. A *release* transition in a contract automaton is translated to two ECC transitions. The first one, s_{ec} to s'_{ec}, leads to a state where the action is forwarded as in the pattern (1). The second transition, s_{ec} to s'''_{ec}, leads to a state that releases an action from a buffer.

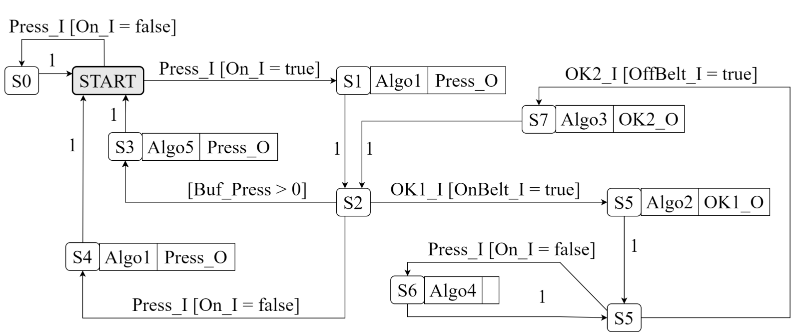
It is not necessary to construct a pattern for rule f_0 in Table 2 because an action is discarded by default when there exists no transition guard that can be satisfied by this action. Moreover, there is an additional transition outgoing to s''_{ec} in every pattern to allow the translation of multiple transitions with the same target state.

Note that the resulting ECC may exhibit nondeterministic behaviour when multiple transitions outgoing from the same state satisfy pattern (5). For instance, two transitions from the same state are guarded with $[Buf\_A > 0]$ and $[Buf\_B > 0]$. In such a case, the ECC would arbitrarily trigger one of the transitions if both buffers contain actions. As an alternative, the techniques could allow the user to give priorities by annotating the *buffer* transitions to specify which one is more important to be released. However, this would make the specification process to become more intricate. Other possibilities are prioritising the *oldest* actions, as in the *First In, First Out* concept, or releasing the longest sequence of actions, as proposed in [24]. All these methods could be implemented, but they would overly complicate the translation patterns, which may cause the ECC to suffer from the state space explosion problem [14].

Example. The synthesised ECC and algorithms of `EnforcerA` from Automaton A are presented in Figure 12. In state *START*, there is an outgoing transition guarded with $Press\_I[ON\_I = true]$ leading to *S0* where no action is executed (i.e., no algorithm nor output event interface). This corresponds to the *discard* transition in state 0 of the automaton. In state *S5*, there is an outgoing transition state *S6* where interface *Press_I* can receive an event when the value of $On\_I = false$. There is no output event interface on *S6*, but there is an algorithm to store the information of the buffered action. Then, the unguarded transition goes back to *S5*. This loop results from a self-loop *buffer* transition in state 2 of Automaton A. The two transitions from *S2* to *S3* and *S4* are synthesised according to the pattern (5) in Table 3. These transitions mean that the action $Press(\{(On, false)\})$ can either be released when available in the buffer (transition *S2* to *S3*) or forwarded when received by the enforcer's input interfaces (transition *S2* to *S4*).

Table 3. Translation patterns

Pattern	Transition in the contract automaton	Corresponding ECC
(1)		
(2)		
(3)		
(4)		
(5)		



(a) ECC

Algo1
1 | On_O := On_I;

Algo2
1 | OnBelt_O := OnBelt_I;

Algo3
1 | OffBelt_O := OffBelt_I;

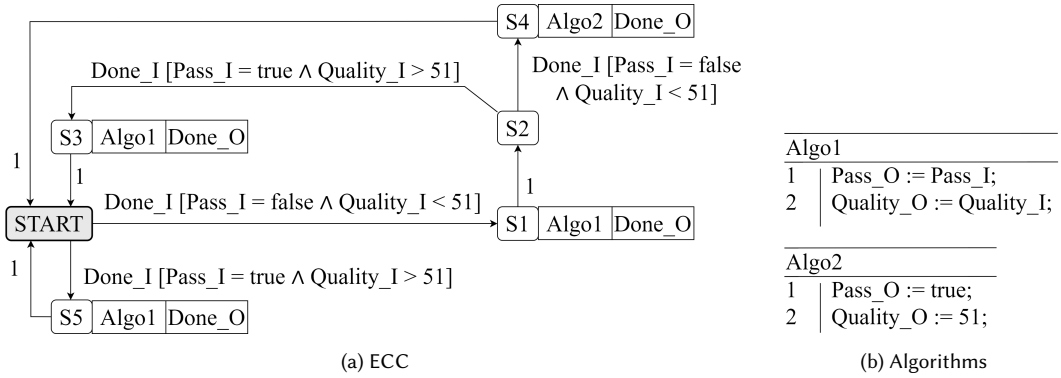
Algo4
1 | Buf_On[Buf_Press] := On_I;
2 | Buf_Press := Buf_Press + 1;

Algo5
1 | On_O := Buf_On[Buf_Press];
2 | Buf_Press := Buf_Press - 1;

(b) Algorithms

Fig. 12. ECC and algorithms of `EnforcerA` FB

The result of the translation from Automaton B to the ECC of `EnforcerB` is depicted in Figure 13. From the starting state of the automaton, there are two transitions typed as *forward*. These are translated as transitions in ECC targeted to states *S1* and *S5*, where the values of data *Pass* and *Quality* are forwarded by executing *Algo1*, and the event *Done* is forwarded from interface *Done_O*. The *replace* transition from 1 to 0 in the contract automaton is translated to ECC's



(a) ECC (b) Algorithms

Fig. 13. ECC and algorithms of `EnforcerB` FB

transition from *S2* to *S4*. Here, when an action satisfies the ECC's transition guard, in state *S4*, the output event of this action is forwarded, and the data is modified by executing *Algo2*.

6.3 Integration

Integrating an enforcer into the application is done by disconnecting the initial connections between the interfaces and creating new connections that include the enforcer.

Algorithm 1: Integration of enforcer

Inputs : $fb_e = (itf_e = (EI_e, EO_e, DI_e, DO_e, WI_e, WO_e), Ag_e, DV_e, ecc_e)$, $iec = (FB, EC, DC)$

Output : $iec' = (FB', EC', DC')$

$FB' := FB \cup \{fb_e\}$ /* enforcer is added to the set of FBs */

EC^{new}, DC^{new}, EC^{old}, $DC^{old} := \emptyset$ /* temporary sets of connections */

foreach $(ei, eo) \in EC$ **do**
  foreach $ei_e \in EI_e$ **do**
    if $eo \approx ei_e$ **then**
      $EC^{old} := EC^{old} \cup \{(ei, eo)\}$ /* an event connection to be removed */
      $EC^{new} := EC^{new} \cup \{(eo, ei_e)\}$ /* an event connection to be added */
  foreach $eo_e \in EO_e$ **do**
    if $ei \approx eo_e$ **then** $EC^{new} := EC^{new} \cup \{(eo_e, eo)\}$
foreach $(di, do) \in DC$ **do**
  foreach $di_e \in DI_e$ **do**
    if $do.id \approx di_e.id$ **then**
      $DC^{old} := DC^{old} \cup \{(di, do)\}$ /* data connection to be removed */
      $DC^{new} := DC^{new} \cup \{(do, di_e)\}$ /* data connection to be added */
  foreach $do_e \in DO_e$ **do**
    if $di.id \approx do_e.id$ **then** $DC^{new} := DC^{new} \cup \{(do_e, di)\}$
$EC' := (EC \setminus EC^{old}) \cup EC^{new}$ /* new connections between interfaces
$DC' := (DC \setminus DC^{old}) \cup DC^{new}$ in the application after the enforcer integration */

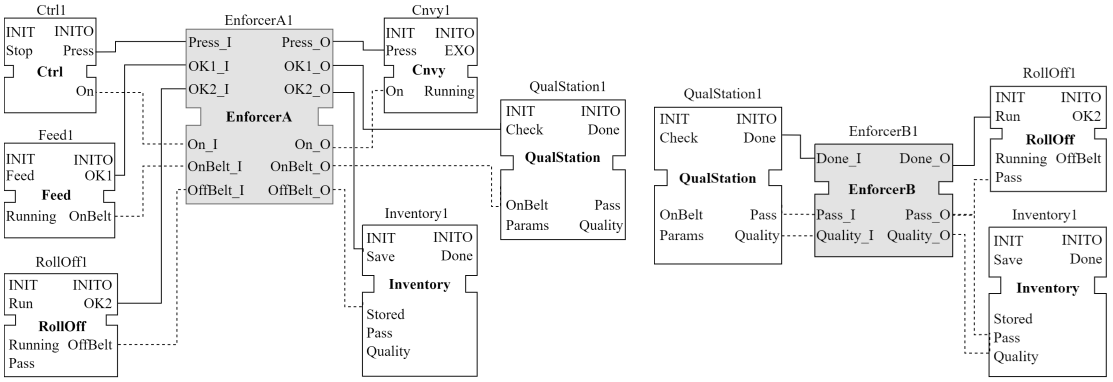


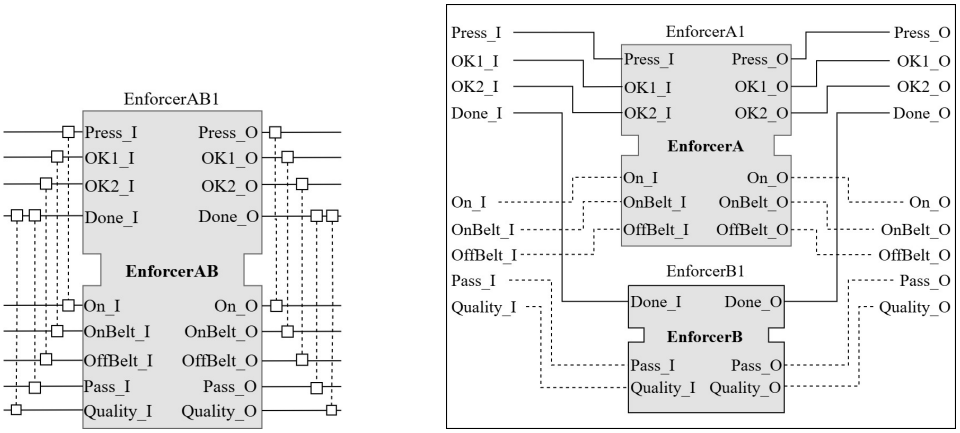
Fig. 14. Enforcers integrated into the application

Algorithm 1 describes the enforcer integration. It takes as input an enforcer fb_e and an IEC 61499 application *iec*. The algorithm generates a new application *iec*′ where the enforcer has been integrated. The symbol ≈ means that two interfaces have different names only because one of them is appended with the suffix "*_I*" or "*_O*". For instance, the `EnforcerA` FB interface *Press_I* and the `Ctrl1` FB interface *Press* have different names because *Press_I* is *Press* appended with the suffix "*_I*"; therefore, *Press_I* ≈ *Press*.

First, the enforcer is merged with the initial set of FBs (line 1). Temporary sets are initialised for facilitating the integration; EC^{new} and DC^{new} store the new connections to be added, whereas EC^{old} and EC^{old} store the connections to be removed (line 2). Next, the algorithm iterates through the set of event connections (lines 3 to 12). When two event interfaces are associated with each other (i.e., $eo \approx ei_e$), the initial event connection is stored in EC^{old} (line 6), and the new event connections are stored in DC^{new} (lines 7 and 10). The iteration through the set of data connections in lines 13 to 22 performs the same task as in event connections. It finds and stores data connections to be removed and added in, respectively, DC^{old} and DC^{new}. Finally, the new sets of event and data connections are created in lines 23 and 24.

Algorithm 1 is correct if it results in sets of event and data interfaces that (i) contain new connections between the enforcer and the existing FBs, (ii) do not contain former connections that must be removed due to the enforcer addition, and (iii) contain former connections that need to be preserved. The first and second points hold because new and former connections are added and removed every time an interface of the enforcer is associated with the interface of an existing FB (lines 5, 10, 15 and 20). The third point holds because no connection is removed except the ones in EC^{old} and DC^{old} (lines 23 and 24).

Example. In order to illustrate the algorithm, let us consider the integrations of `EnforcerA` and `EnforcerB` into the IEC 61499 application described in Section 3. The results are depicted in Figure 14. Only new connections between the enforcers and the existing FBs are shown for clarity. The other connections remain the same as presented in Figure 5 (b). As described in Algorithm 1, event connections in the application associated with the enforcer's interfaces are removed, followed by the creation of new connections. For instance, to integrate `EnforcerA`, *Press* output event interface in `Ctrl1` was initially connected to *Press* input event interface in `Cnvy1`. This connection is removed, and new connections are created between *Press* in `Ctrl1` to *Press_I* and *Press_O* to *Press* in `Cnvy1`. Similarly, data connections between *Pass* in



(a) Interfaces (b) Enforcers

Fig. 15. Example of composite enforcer

`QualStation1` to *Pass* in `RollOff1` and `Inventory1` are replaced with connections from *Pass* to *Pass_I*. Then, *Pass_O* is connected to *Pass* in both `RollOff1` and `Inventory1`.

Note that when an enforcer is to be connected with FBs from different sub-applications, then service interface FBs such as `PUBLISH` and `SUBSCRIBE` are required to send events and data between the sub-applications. Suppose `Ctrl1` FB in Figure 14 (a) is in a different sub-application. In this case, a `PUBLISH` FB must be added to that sub-application and connected with the interfaces of `Ctrl1`. Also, a `SUBSCRIBE` FB must be added and connected with `EnforcerA1` [2].

6.4 Composite enforcer

Several adaptations may occur during the execution of an IEC 61499 application. As a result, the application can contain multiple enforcers, which may make it difficult for developers to keep track of and maintain all the added enforcers. In order to mitigate this problem, we propose a method to merge enforcers into one composite enforcer.

A composite enforcer is a composite FB consisting of several enforcers. Its interfaces are built from the union of all existing event and data interfaces. There is no connection between the constituent enforcers because they are independent. Multiple actions can be modified in parallel if each corresponds to a unique enforcer inside the composite enforcer [56].

Definition 6.4. (Composite enforcer) A composite enforcer is a tuple (FB_c, itf_c), where $FB_c = \{fb_e^1, fb_e^2, ..., fb_e^n\}$ is a set of enforcers and $itf_c = \bigcup_{i=1}^{n} fb_e^i.itf_e$ is the interfaces.

Example. An example of a composite enforcer consisting of `EnforcerA` and `EnforcerB` is shown in Figure 15. The interfaces in Figure 15 (a) are the union of both enforcers' interfaces. For instance, *Press_I* is an input event interface from `EnforcerA`, whereas *Quality_O* is an output data interface from `EnforcerB`. In Figure 15 (b), both enforcers are assembled inside the composite FB without any connection between them. To integrate a composite enforcer, we first remove the selected enforcers (e.g., `EnforcerA` and `EnforcerB`) from the application. Then, Algorithm 1 is applied to connect the composite enforcer with the other FBs.

[2]More details on how to use `PUBLISH` and `SUBSCRIBE` FBs can be found on the 4DIAC-IDE documentation webpage.

6.5 Characteristics

The runtime approach for IEC 61499 must be proven correct to use it for adapting ICSs reliably according to requirements. According to [24, 34, 47], the correctness of a runtime enforcement approach is proven when it satisfies soundness, transparency, and deadlock-freedom characteristics.

Soundness is satisfied when the application executes according to the requirements. Our approach must be sound because the enforcers should modify the execution of the application such that the requirements are satisfied. The runtime enforcement techniques are transparent when they do not induce unexpected behaviour. Transparency is needed because the enforcers should not make any unspecified modifications to the application execution. Deadlock-freedom is satisfied when the application can always make progress. It is required because the enforcers should not make the application stop executing. In the following sections, we explain why soundness and transparency are guaranteed by construction. Afterwards, the use of model checking to ensure deadlock-freedom is described.

6.5.1 Soundness and transparency. Suppose that a set of enforcers FB_e are synthesised from a set of contract automata C. These enforcers are integrated into an initial IEC 61499 application $iec = (FB, EC, DC)$, such that it becomes $iec' = (FB \cup FB_e, EC', DC')$.

Proposition 1 states that every trace produced by the application satisfies the contract automata.

Proposition 1. ***(Soundness).*** $\forall \sigma \in Run(iec') : \sigma \vDash C$

Proof (sketch). Soundness is satisfied because the enforcers execute every possible modification specified by the contract automata. By construction, the enforcers' ECCs define the behaviours that respect the transition rules in Table 2 by mapping every translation pattern in Table 3 into each rule. These mappings from rules to patterns are as follows: *f1* to (1), *d1* to (2), *r1* to (3), *b1* to (4), and *f2* to (5).

Proposition 2 states that the enforcers do not modify a trace satisfying the contract automata.

Proposition 2. ***(Transparency).*** $\forall \sigma \in Run(iec) : \sigma \vDash c \Rightarrow \sigma \in Run(iec')$

Proof (sketch). The enforcers modify the application execution only when that modification is specified in the contract automata. When an action is not associated with any guard in the contract automata, then the synthesised enforcers do not provide the interfaces corresponding to this action. This implies that it is impossible for the enforcers to modify this action.

6.5.2 Deadlock-freedom. Deadlock-freedom is not guaranteed by construction because the user may specify contract automata synthesised into enforcers that cause deadlock (e.g., by discarding events required for making progress). Therefore, we propose to use model checking techniques to ensure deadlock-freedom.

To apply model checking, the IEC 61499 application and the synthesised enforcers are first transformed into a behavioural model. This model can be obtained using the translation method proposed in [21]. In this method, the application and the enforcers are translated into an LNT specification [13], which is then compiled into a Labelled Transition System (LTS) model [31] consisting of states and transitions labelled with actions. In order to check that an LTS model is free from deadlock, we must specify a property stating that every state has at least one successor. This property is written in MCL [38] as $[true*] < true > true$. The CADP model checker [27] is then used to check the property on the LTS model. If *true* is returned, then deadlock-freedom is guaranteed. Note that our approach for ensuring deadlock-freedom is generic. There exist other works in [6, 19, 39] that can also be used to apply model checking techniques for IEC 61499 using various formal languages and model checkers.

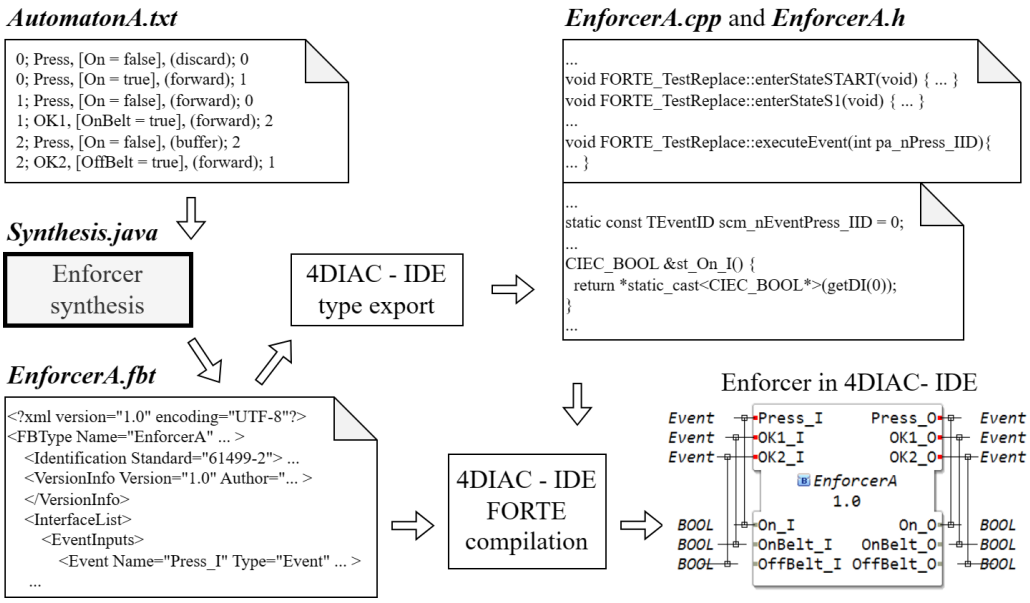


Fig. 16. Implementation of enforcer synthesis

7 Implementation

This section presents the implementation of the IEC 61499 runtime enforcement approach. We first focus on the technical details of the developed tool for synthesising enforcers. Afterwards, experimental results are presented to show that enforcers do not induce performance overhead. Finally, we discuss threats to the validity of the results.

Our implementation relies on 4DIAC-IDE [54], an open-source development tool for IEC 61499 applications. The developed tools and experiments presented in this section are available in an online repository [26]. This repository also contains more examples for readers interested in them. One of them is a more complex conveyor test station containing 23 composite FBs with more than 100 FBs inside of them in total. There is also a temperature control system with an enforcer that consists of 32 states and 44 transitions.

7.1 Tool Support

The technical detail for synthesising enforcers is presented in Figure 16. A program written in the Java programming language has been developed to implement the synthesis techniques described in Section 6.2. It returns an enforcer (i.e., a basic FB) in the form of an XML file for a given contract automaton, which is written in a text file. 4DIAC-IDE is used to read the generated enforcers. Every new enforcer must be exported and compiled together with the runtime environment called FORTE [3]. Once all the steps are completed, the enforcer can be integrated into the application.

Note that the maintainability of the generated enforcers is not an issue because each enforcer is computed automatically using the synthesis techniques and guaranteed to behave according to the specified contract automaton. Enforcers are transparent to the users. More precisely, users can use 4DIAC-IDE if they wish to see the internal elements of an

[3]This compilation process is explained on the official webpage of 4DIAC-IDE.

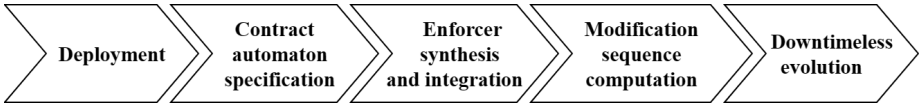


Fig. 17. Adaptation of physical systems using runtime enforcement for IEC 61499

enforcer, such as its ECC. It is also possible to use the FORTE runtime environment to simulate and observe the enforcer behaviour by triggering its input event interfaces.

Figure 17 shows the steps required to use our approach for adapting physical systems. The first step is to deploy the application. 4DIAC-IDE provides a comprehensive guide about the deployment of IEC 61499 applications. Once the application is running, a contract automaton is specified to be synthesised as an enforcer. Algorithm 1 computes a target application in which the enforcer has been integrated. Next, a modification sequence is computed from the initial application (without an enforcer) and the target application. This sequence is a list of operations required for preserving dependencies during the dynamic modification (e.g., the connections of an FB must be removed before deleting that FB). The work in [46] describes the automated computation of the modification sequence. Finally, the initial application is modified into the target application according to the modification sequence without stopping the physical system. Existing tools and frameworks, such as the one proposed in [41], can be used to complete this task.

7.2 Experiments

The main purpose of these experiments is to compare the performance of IEC 61499 applications before and after the integration of enforcers. This is done to make sure that the enforcers do not induce significant performance overhead. We take several real-world examples of IEC 61499 applications from the existing literature. Requirements are then specified using contract automata. These automata are synthesised as enforcers and integrated into the corresponding applications. Finally, we execute multiple simulations of these applications to obtain the average execution time with and without enforcers. These simulations are performed using 4DIAC-IDE and FORTE runtime environment.

The results of our experiments are presented in Table 4. The first two columns contain the application names and the numbers of FBs representing the size of the application. The next two columns contain the number of states and transitions in the contract automata to represent the complexity of each contract automaton. The fifth and sixth columns contain the number of states and transitions of the enforcers' ECCs, whereas the seventh and eighth columns contain the number of event and data interfaces of the enforcers. They represent the complexity of the synthesised enforcer. The last two columns contain the average execution time before and after enforcers' integrations in milliseconds. Note that some of the experiments were performed using the same applications but different contract automata. Therefore, they are named differently (e.g., applications Blinking A and Blinking B).

The results in Table 4 help us to answer the second research question introduced in Section 1 about the applicability of our approach in terms of performance. In most cases, there is no significant overhead when an enforcer is integrated into the application. Experiment 4 resulted in the highest increase of execution time (23 ms). This is because, in this specific experiment, multiple complex enforcers were integrated into an application that consists of many FBs. However, the increase is much less than a second, thus negligible. Therefore, we may conclude that our approach is applicable in terms of performance.

Table 4. Experimental results

Application		CA		ECC		Itf.		ET(ms)	
Name	FBs	St.	Tr.	St.	Tr.	Ev.	Da.	Be.	Af.
1. Conveyor 1-A [68]	7	3	6	10	14	6	6	59167	59108
2. Conveyor 1-B [68]	7	2	4	6	7	2	4	59167	59151
3. Conveyor 2-A [68]	23	3	6	10	14	6	6	83106	83115
4. Conveyor 2-C [68]	23	10	20	30	40	2	4	83106	83129
5. Temperature A [68]	4	2	4	6	8	2	2	10005	10003
6. Temperature B [68]	4	5	12	18	24	2	2	10005	10004
7. Temperature 2-C [68]	8	11	22	32	44	2	2	10003	10005
8. Capping A [21]	5	2	4	6	8	4	0	10005	10004
9. Capping B [21]	5	5	12	18	24	4	0	10005	10003
10. Blinking A [1]	3	2	2	4	4	4	0	10002	10003
11. Blinking B [1]	3	2	4	6	8	4	0	10002	10005
Average								32233	32230

7.3 Threats to Validity

The experimental results are obtained using simulations in 4DIAC-IDE. There are two potential threats to the validity of these results. The first one concerns the time intervals between the events in the simulations that may not reflect those in physical systems. In physical systems, some events are triggered due to the interactions between the sensors and the environment. In our simulations, these events were created using additional FBs that can trigger events periodically called `E_CYCLE`. Nevertheless, there should not be a significant threat to the validity of the results in this regard because the execution time of the enforcer does not depend on the execution time of other FBs.

Secondly, our experiments do not consider distributed applications. In such applications, there may be additional overheads when the enforcers must receive and send events through the network. However, recent studies show that the overheads caused by network communication in IEC 61499 applications are negligible [7, 37]. Hence, this aspect does not pose a major threat to the validity of the results.

8 Related Work

This section compares our approach with other related works from five different perspectives. Section 8.1 discusses the differences between contract automata and existing specification languages for runtime enforcement. Section 8.2 surveys existing techniques to support adaptable IEC 61499 applications. Sections 8.3 and 8.4 describe the works on formal verification and runtime enforcement for ICSs, respectively. Section 8.5 explains existing works on self-adaptive systems.

8.1 Specification Language

Security Automata (SA), introduced in [49], is one of the earliest runtime enforcement specification languages. It is an FSM-based language that runs in parallel with the system. A security automaton can specify to halt the system when a transition corresponding to a certain action is not available in the current state. Halting is not an option in our case because the control system must continue making progress. For instance, the industrial materials would be congested if the conveyor test station (Section 3) is halted. In comparison, our contract automata can specify to discard, replace, or buffer the undesired action without stopping the IEC 61499 application.

Edit Automata (EA) described in [36] is a well-known language to enforce system properties. It extends SA's capability to insert and suppress actions. This feature is called buffering in contract automata. In addition to that, contract automata can discard or replace actions. The discard mechanism can be useful when it is not necessary to buffer certain actions. For instance, when a conveyor is switched on, discarding the action that corresponds to switching on the conveyor is preferred over buffering the action.

A specification language based on temporal logic [43] called Event-Driven Temporal Logic (EDTL) is introduced in [70]. The language can specify requirements based on six unique attributes. For instance, *trigger* attribute describes the starting event of the requirement, while *reaction* attribute is associated with the ending event. In comparison, we use different types of transition in contract automata to specify the desired application behaviour.

8.2 Adaptable IEC 61499 Applications

The work in [50] describes scalable reconfiguration techniques for IEC 61499 applications. Given an application and requirements expressed using first-order formulae, a reconfigured application satisfying the requirements is computed using SMT analysis methods. This work deals with structural problems of the application, such as there should only be one copy of a certain FB. In contrast, our approach enforces a desired system execution. For instance, two events that can be triggered from a specific FB must alternate. Furthermore, the target application in their approach may involve several additions/removals of FBs, whereas in our approach, a single FB is added.

A collection of works in [44–46] proposes techniques to resolve the dynamic reconfiguration of IEC 61499 applications. The initial approach in [46] relies on a predefined set of rules to build dependency trees for given source and target applications. A dependency tree is used to build the sequence of dynamic reconfiguration. In [44], the authors analyse the execution time of these reconfigurations in various scenarios. Finally, the work in [45] describes the techniques to generate rollback sequences when reconfigurations of applications need to be cancelled. These works aim at finding reconfiguration sequences that can be applied while the application is running. Such reconfigurations may involve additions or removals of multiple FBs. Also, there is no notion of requirements. The target application given by the user is assumed to be a correct application. In comparison with our work, a user can formulate the requirements as a contract automaton, and the only modification is the integration of an enforcer synthesised from this automaton.

The authors in [52, 53, 65] propose to support the adaptability of IEC 61499 applications using specific design patterns. The earliest one in [65] studies distributed hierarchical design patterns, which can be used when the application involves a lot of parallelism. In [53], this pattern is extended to further support the reconfigurability and reusability by differentiating FBs based on their interactions. The most recent one [52] introduces new FBs to the pattern dedicated to error detection and handling. These works guide the users in building adaptable IEC 61499 applications using design patterns. In contrast, our goal is to apply a minimal change to the application to satisfy specific requirements.

The authors in [2] propose to support adaptive ICSs by defining the formal notion of *explanation*. Explanations contain information about certain adaptation decisions. Each explanation is characterised by content, effect, and cost. Probabilistic model checking is used to obtain the optimal explanation, which allows human operators to understand the change in system behaviours during an adaptation process. When a questionable adaptation occurs, the operator may cancel the process. Compared to our techniques, this work also involves users as *human-in-the-loop*. In our work, users can specify requirements to be adapted by synthesising and integrating an enforcer to change the runtime execution of the system, whereas their work helps the users to understand adaptation processes using the generated explanations.

8.3 Formal Verification

Our approach is within the scope of formal verification. This section shows that existing works on IEC 61499 focus on verifying the application's correctness (e.g., using model checking) according to the requirements (i.e., properties). In comparison, our approach aims to modify the application execution at runtime to satisfy the requirements.

The works in [6, 19, 30, 32] apply model checking to verify control programs developed using various languages. They also use different types of temporal logic formalisms to express the properties. The authors in [30] focus on systems written in Sequential Function Chart (SFC). Here, safety properties such as deadlock freedom were identified using a toolchain called SAVE/SFC. The approach proposed in [32] targets PLC programs developed using the IEC 61131-3 standard. The correctness of these programs is checked using the SMV model checker, and the method is proven to be consistent by matching the temporal logic formulas to the languages used in the standard (i.e., Structured Text and SFC). The work in [19] aims at verifying IEC 61499 applications by also taking into account the time constraint. In this regard, the approach is suitable for ensuring the correctness of applications that involve FBs associated with time, such as *E_DELAY*. In [6], basic FBs are extended into so-called *safety* FBs, which can describe qualitative and quantitative properties. The application is then translated into a PRISM model to be verified using the PRISM model checker. Model checking, which is used in these works, is a well-established method to verify systems' correctness during design time. Complementarily, the focus of our work is to support control systems when requirements must be satisfied after the deployment.

In [21, 25], the authors combine static and runtime aspects for verifying IEC 61499 applications. An application is first translated into a specification language called LNT using translation patterns. Afterwards, a behavioural model called LTS is generated from this specification. The model is then enriched using execution traces obtained from monitoring techniques. As a result, a probabilistic model called PTS is produced. This final model is used to check probabilistic properties given by the user. This work involves the analysis of the application's dynamic behaviour by inserting an FB to monitor the execution. In comparison, we propose to integrate an FB called enforcer that can modify the application's execution according to a specification.

8.4 Runtime Enforcement

The authors in [42] propose runtime enforcement techniques for cyber-physical systems, in particular, the heart pacemaker system. The requirements are expressed as Synchronous Discrete Time Automata (SDTA). This language allows, for instance, to forbid two different pace signals (i.e., *atrial* and *ventricular* paces) to happen simultaneously. In comparison with our work, the states of SDTA are distinguished into two types: accepting and non-accepting. A contract automaton does not require this distinction because it aims only at specifying how the system can achieve a desired behaviour (as opposed to undesired behaviour where non-accepting states are necessary). In SDTA, when the transition goes to a non-accepting state, the corresponding action is replaced with another existing action on a transition outgoing

from the same source state. This is similar to our *replace* mechanism, where the original and replacement actions are specified on the same transition. Their approach also does not halt the system because the heart pacemaker is a reactive system in which it must trigger an output every time there is an input. Hence, buffering is not an option for them, whereas it is acceptable for us because the buffered action can be released in the next execution. Furthermore, their enforcer is bi-directional because it receives inputs and sends outputs from the heart and the pacemaker. In contrast, our enforcer is unidirectional, but its input and output interfaces can be connected to multiple FBs.

The work in [34] applies runtime enforcement to industrial control systems relying on the IEC 61131-3 standard [16]. The controller programs are expressed using a language based on a timed process calculus, while the correctness of these programs is specified using time correctness properties. Similarly to ours, a property is synthesised as an enforcer. The properties that can be specified in this work are specific, such as *bounded eventually*, *bounded absence*, and *bounded maximum duration*. For comparison, we propose to express properties using contract automata. This allows the users to specify requirements to be enforced by using different types of transitions. Moreover, their work targets the IEC 61131-3 standard, where programs execute using a scan-based model, whereas ours focuses on IEC 61499, which relies on an event-driven execution model.

This paper revises and significantly extends the previous work in [22] with several new contributions. First, we introduce a new input language called contract automata, which is more expressive than the previous one (known as property automata). The enhanced expressiveness comes from the new type of transition called *buffer*. This type allows the user to specify certain actions to be buffered and released in future execution. In this work, we also perform simulations of several realistic IEC 61499 applications. The results are observed to ensure that enforcers do not induce significant performance overhead. This paper also presents (i) a formal model of IEC 61499 for runtime enforcement, (ii) formal semantics of the contract automata, and (iii) translation patterns to transform a contract automaton into an ECC and an algorithm to integrate enforcers. In addition, a new tool support was developed to automatically synthesise enforcers from contract automata.

8.5 Self-Adaptive Systems

The work in [8] introduces a goal modelling notation called EDGE to support self-adaptive systems. This notation is proposed based on five *desiderata* (i.e., requirements), such as automated goal selection. EDGE models are synthesised into goal controllers using the PRISM model checker [33]. The goal controllers in this work are comparable with enforcers in our work. However, a goal controller aims to change the system's objective. For instance, it can make the conveyor test station perform a new task, such as repairing defective materials. In comparison, an enforcer focuses on changing the system's behaviour, such as making the conveyor test station accept every material.

Model checking of stochastic multiplayer games is applied in [10] for analysing latency-aware adaptations. An algorithm is proposed to choose optimal adaptations at runtime using the latency information. Simulations on a content provider website show that the approach can help to balance the system's response time, the number of active servers, and the content fidelity level. These techniques focus on adaptation to optimise the non-functional aspects of the system, such as response time. In contrast, our approach aims at behavioural aspects of the system. It allows the system to adapt to requirements, such as the conveyor should stop running after a certain sequence of actions.

An end-to-end framework called ENTRUST dedicated to self-adaptive systems is proposed in [9]. This framework can generate *assurance cases*, which guarantee the trustworthiness of adaptations. Case studies on underwater vehicles and foreign exchange trade domains show positive results. For instance, ENTRUST can make correct adaptation decisions with valid assurance cases. This work defines a methodology to develop trustworthy self-adaptive systems from design

time to runtime (end-to-end). In comparison, our method can be applied during runtime when the system needs to adapt to certain requirements. Furthermore, they use assurance cases to prove that adaptations are correct, whereas we prove the correctness by showing that our runtime enforcement approach is sound and transparent.

The authors in [57] propose an executable modelling language called EUREMA to develop adaptation engines. The language supports useful features, such as allowing multiple feedback loops and structural adaptation. An instance of a specification written in EUREMA is called a feedback loop diagram. Comparatively, the work in [64] presents ActivFORMS, a framework for also designing feedback loops in self-adaptive systems. There are four stages in the development: design, deployment, runtime adaptation, and evolution. Specific techniques are applied in every stage to ensure the feedback loop correctness efficiently. For instance, statistical model checking is used at runtime to help select adaptation options correctly and also efficiently because it avoids exploring the whole state space. These two works allow users to write various adaptation scenarios using feedback loop diagrams, which are very expressive. A diagram contains processes, models, and control flows with different types of states and transitions. In our approach, requirements are expressed using contract automata, which focuses on behavioural adaptations.

9 Concluding Remarks

This paper has presented an approach to support adaptive industrial control systems using the IEC 61499 standard and runtime enforcement techniques. The idea is to make IEC-61499-based applications adapt to specific requirements by modifying their runtime execution. Tedious tasks such as replacements of FBs and adjustments of FB connections can thus be avoided.

The method starts with an expressive and compact input specification called contract automaton. This automaton allows the user to specify different modifications of actions according to the requirements. An action can be forwarded, discarded, replaced, or buffered. Next, the automaton is transformed into an enforcer in the form of a basic FB. After the integration, this enforcer can modify the application execution. A tool was developed to automate the synthesis of enforcers. Experiments were presented to show that the enforcer does not induce significant performance overhead.

For future work, we envision a self-adaptive industrial control system relying on the IEC 61499 standard and runtime enforcement techniques. This could be achieved by automatically generating the requirements to be satisfied. These requirements could be identified by monitoring the system itself and its environment. We would then design a decision framework to determine when the system should be adapted to the identified requirements. This framework must also be able to construct a contract automaton according to the requirements. Finally, we could apply the proposed synthesis and integration of enforcer methods so that the system can adapt to the requirements without human intervention.

Acknowledgments

This work is supported by the French National Research Agency in the framework of the « France 2030 » program (ANR-15-IDEX-0002) and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

References

[1] 4DIAC. 2024. 4DIAC Documentation. https://www.eclipse.org/4diac/en_help.php?helppage=html/4diacIDE/use4diacLocally.html.

[2] Sridhar Adepu, Nianyu Li, Eunsuk Kang, and David Garlan. 2022. Modeling and Analysis of Explanation for Secure Industrial Control Systems. *ACM Trans. Auton. Adapt. Syst.* 17, 3-4 (2022), 1–26. https://doi.org/10.1145/3557898

[3] ANSI/IEEE. 1983. IEEE Standard Glossary of Software Engineering Terminology. *ANSI/IEEE Std 729-1983* (1983), 1–40. https://doi.org/10.1109/IEEESTD.1983.7435207

[4] Tom Mejer Antonsen. 2020. *PLC Controls with Structured Text (ST), V3 Monochrome: IEC 61131-3 and best practice ST programming*. Books on Demand.

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

[6] Zeeshan Ejaz Bhatti, Partha S. Roop, and Roopak Sinha. 2017. Unified Functional Safety Assessment of Industrial Automation Systems. *IEEE Trans. Ind. Informatics* 13, 1 (2017), 17–26. https://doi.org/10.1109/TII.2016.2610185

[7] Friederike Bruns, Bianca Wiesmayr, and Alois Zoitl. 2023. Supporting Model-Based Network Specification for Time-Critical Distributed Control Systems in IEC 61499. In *Proc. of CASE'23*. IEEE, 1–7. https://doi.org/10.1109/CASE56687.2023.10260604

[8] Radu Calinescu and Genaína Nunes Rodrigues. 2023. Goal Controller Synthesis for Self-Adaptive Systems. In *Proc. of FormaliSE'23*. IEEE, 1–6. https://doi.org/10.1109/FORMALISE58978.2023.00008

[9] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly. 2018. Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Trans. Software Eng.* 44, 11 (2018), 1039–1069. https://doi.org/10.1109/TSE.2017.2738640

[10] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley R. Schmerl. 2016. Analyzing Latency-Aware Self-Adaptation Using Stochastic Games and Simulations. *ACM Trans. Auton. Adapt. Syst.* 10, 4 (2016), 23:1–23:28. https://doi.org/10.1145/2774222

[11] Javier Cámara, Javier Troya, Antonio Vallecillo, Nelly Bencomo, Radu Calinescu, Betty H. C. Cheng, David Garlan, and Bradley R. Schmerl. 2022. The uncertainty interaction problem in self-adaptive systems. *Softw. Syst. Model.* 21, 4 (2022), 1277–1294. https://doi.org/10.1007/S10270-022-01037-6

[12] Stefano Campanelli, Pierfrancesco Foglia, and Cosimo Antonio Prete. 2012. Integration of existing IEC 61131-3 systems in an IEC 61499 distributed solution. In *Proc. of ETFA'12*. IEEE, 1–8. https://doi.org/10.1109/ETFA.2012.6489671

[13] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. 2024. Reference Manual of the LNT to LOTOS Translator (Version 7.3). (2024). INRIA/VASY and INRIA/CONVECS, 148 pages.

[14] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. 2011. Model Checking and the State Explosion Problem. In *Proc. of LASER'11 (LNCS, Vol. 7682)*. Springer, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1

[15] International Electrotechnical Commission. 2012. Functional blocks - Part 1: Architecture. *IEC 61499-1* (2012).

[16] International Electrotechnical Commission. 2013. Programmable controllers-part 3: Programming languages. *IEC 61131-3* (2013).

[17] Wenbin William Dai and Valeriy Vyatkin. 2009. A case study on migration from IEC 61131 PLC to IEC 61499 function block control. In *Proc.of INDIN'09*. IEEE, 79–84. https://doi.org/10.1109/INDIN.2009.5195782

[18] Dmitrii Drozdov, Victor Dubinin, Sandeep Patil, and Valeriy Vyatkin. 2021. A Formal Model of IEC 61499-Based Industrial Automation Architecture Supporting Time-Aware Computations. *IEEE Open Journal of the Ind. Electronics Society* 2 (2021), 169–183. https://doi.org/10.1109/OJIES.2021.3056400

[19] Dmitrii Drozdov, Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. 2016. Formal verification of cyber-physical automation systems modelled with timed block diagrams. In *Proc. of ISIE'16*. IEEE, 316–321. https://doi.org/10.1109/ISIE.2016.7744910

[20] Yliès Falcone. 2010. You Should Better Enforce Than Verify. In *Proc. of RV'10 (LNCS, Vol. 6418)*. Springer, 89–105. https://doi.org/10.1007/978-3-642-16612-9_9

[21] Yliès Falcone, Irman Faqrizal, and Gwen Salaün. 2022. Probabilistic Analysis of Industrial IoT Applications. In *Proc. of IoT'22*. ACM, 41–48. https://doi.org/10.1145/3567445.3567461

[22] Yliès Falcone, Irman Faqrizal, and Gwen Salaün. 2022. Runtime Enforcement for IEC 61499 Applications. In *Proc. of SEFM'22 (LNCS, Vol. 13550)*. Springer, 352–368. https://doi.org/10.1007/978-3-031-17108-6_22

[23] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. 2018. Runtime Failure Prevention and Reaction. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. LNCS, Vol. 10457. Springer, 103–134. https://doi.org/10.1007/978-3-319-75632-5_4

[24] Yliès Falcone and Gwen Salaün. 2021. Runtime Enforcement with Reordering, Healing, and Suppression. In *Proc. of SEFM'21 (LNCS, Vol. 13085)*. Springer, 47–65. https://doi.org/10.1007/978-3-030-92124-8_3

[25] Irman Faqrizal, Tatiana Liakh, Midhun Xavier, Gwen Salaün, and Valeriy Vyatkin. 2024. Probabilistic Model Checking for IEC 61499: A Manufacturing Application. In *Proc. of ICIT'24*. IEEE, 1–6. https://doi.org/10.1109/ICIT58233.2024.10540845

[26] Runtime Enforcement for IEC 61499. 2024. Tool Support. https://gitlab.inria.fr/ifaqriza/runtime-enforcement-for-iec-61499.

[27] Hubert Garavel, Frédéric Lang, Radu Mateescu, et al. 2013. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT* 15, 2 (2013), 89–107.

[28] Martin Glinz. 2007. On Non-Functional Requirements. In *Proc. of RE'07*. IEEE Computer Society, 21–26. https://doi.org/10.1109/RE.2007.45

[29] Peter Gsellmann, Martin Melik-Merkumians, Alois Zoitl, and Georg Schitter. 2021. A Novel Approach for Integrating IEC 61131-3 Engineering and Execution Into IEC 61499. *IEEE Trans. Ind. Informatics* 17, 8 (2021), 5411–5418. https://doi.org/10.1109/TII.2020.3033330

[30] H. Kawata and N. Uchihira. 1996. Practical program validation for plant control systems using SFC and temporal logic. In *Proc. of SMC'96*, Vol. 4. 3186–3191. https://doi.org/10.1109/ICSMC.1996.561496

[31] Robert M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (1976), 371–384. https://doi.org/10.1145/360248.360251

[32] E. V. Kuzmin, D. A. Ryabukhin, and Valery A. Sokolov. 2016. On the expressiveness of the approach to constructing PLC-programs by LTL-specification. *Autom. Control. Comput. Sci.* 50, 7 (2016), 510–519. https://doi.org/10.3103/S0146411616070130

[33] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2002. PRISM: Probabilistic Symbolic Model Checker. In *Proc. of TOOLS'02 (LNCS, Vol. 2324)*. Springer, 200–204. https://doi.org/10.1007/3-540-46029-2_13

[34] Ruggero Lanotte, Massimo Merro, and Andrei Munteanu. 2023. Industrial Control Systems Security via Runtime Enforcement. *ACM Trans. Priv. Secur.* 26, 1 (2023), 4:1–4:41. https://doi.org/10.1145/3546579

[35] Tatiana Liakh, Radimir Sorokin, Daniil Akifev, Sandeep Patil, and Valeriy Vyatkin. 2022. Formal model of IEC 61499 execution trace in FBME IDE. In *Proc. of INDIN'22*. IEEE, 588–593. https://doi.org/10.1109/INDIN51773.2022.9976176

[36] Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.* 4, 1-2 (2005), 2–16. https://doi.org/10.1007/s10207-004-0046-8

[37] Per Lindgren, Johan Eriksson, Marcus Lindner, Andreas Lindner, David Pereira, and Luís Miguel Pinho. 2015. Response time for IEC 61499 over Ethernet. In *Proc. of INDIN'15*. IEEE, 1206–1212. https://doi.org/10.1109/INDIN.2015.7281907

[38] Radu Mateescu and Mihaela Sighireanu. 2003. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.* 46, 3 (2003), 255–281. https://doi.org/10.1016/S0167-6423(02)00094-1

[39] Polina Ovsiannikova and Valeriy Vyatkin. 2021. Towards user-friendly model checking of IEC 61499 systems with counterexample explanation. In *Proc. of ETFA'21*. 01–04. https://doi.org/10.1109/ETFA45728.2021.9613491

[40] Cheng Pang, Wenbin William Dai, and Valeriy Vyatkin. 2015. Towards IEC 61499 models of computation in Ptolemy II. In *Proc. of IECON'15*. IEEE, 1988–1993. https://doi.org/10.1109/IECON.2015.7392392

[41] Eliseu Moura Pereira, João Pedro Correia dos Reis, and Gil Gonçalves. 2020. DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems. In *Proc. of SAM-IoT'20 (CEUR, Vol. 2739)*. CEUR-WS.org, 63–71. https://ceur-ws.org/Vol-2739/paper_9.pdf

[42] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard von Hanxleden. 2017. Runtime Enforcement of Cyber-Physical Systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 178:1–178:25. https://doi.org/10.1145/3126500

[43] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proc. of FOCS'77*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[44] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. 2022. Real-time Dynamic Reconfiguration for IEC 61499. In *Proc. of ICPS'22*. IEEE, 1–6. https://doi.org/10.1109/ICPS51978.2022.9816872

[45] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. 2022. Rollback Sequences for Dynamic Reconfiguration of IEC 61499. In *Proc. of INDIN'22*. IEEE, 81–86. https://doi.org/10.1109/INDIN51773.2022.9976148

[46] Laurin Prenzel and Sebastian Steinhorst. 2021. Automated Dependency Resolution for Dynamic Reconfiguration of IEC 61499. In *Proc. of ETFA'21*. 1–8. https://doi.org/10.1109/ETFA45728.2021.9613156

[47] Matthieu Renard, Antoine Rollet, and Yliès Falcone. 2020. Runtime enforcement of timed properties using games. *Formal Aspects Comput.* 32, 2-3 (2020), 315–360. https://doi.org/10.1007/S00165-020-00515-2

[48] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* 4, 2 (2009), 14:1–14:42. https://doi.org/10.1145/1516533.1516538

[49] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (feb 2000), 30–50. https://doi.org/10.1145/353323.353382

[50] Roopak Sinha, Kenneth Johnson, and Radu Calinescu. 2014. A scalable approach for re-configuring evolving industrial control systems. In *Proc. of ETFA'14*. IEEE, 1–8. https://doi.org/10.1109/ETFA.2014.7005126

[51] Roopak Sinha, Partha S. Roop, Gareth Shaw, Zoran A. Salcic, and Matthew M. Y. Kuo. 2016. Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks. *IEEE Trans. Ind. Informatics* 12, 1 (2016), 59–68. https://doi.org/10.1109/TII.2015.2496262

[52] Lisa Sonnleithner, Bianca Wiesmayr, Virendra Ashiwal, Shubham Sharma, Alois Zoitl, and Jörg Walter. 2022. Architectural Concepts for IEC 61499-based Machine Controls: Beyond Normal Operation Handling. In *Proc. of ETFA'22*. 1–8. https://doi.org/10.1109/ETFA52439.2022.9921610

[53] Lisa Sonnleithner, Bianca Wiesmayr, Virendra Ashiwal, and Alois Zoitl. 2021. IEC 61499 Distributed Design Patterns. In *Proc. of ETFA'21*. IEEE, 1–8. https://doi.org/10.1109/ETFA45728.2021.9613569

[54] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Alois Zoitl, Christoph Sunder, Antonio Valentini, and Allan Martel. 2008. Framework for Distributed Industrial Automation and Control (4DIAC). In *Proc. of INDIN'08*. 283–288. https://doi.org/10.1109/INDIN.2008.4618110

[55] Christoph Sünder, Valeriy Vyatkin, and Alois Zoitl. 2013. Formal Verification of Downtimeless System Evolution in Embedded Automation Controllers. *ACM Trans. Embed. Comput. Syst.* 12, 1 (2013), 17:1–17:17. https://doi.org/10.1145/2406336.2406353

[56] Christoph Sunder, Alois Zoitl, James. H. Christensen, Marco Colla, and Thomas Strasser. 2007. Execution Models for the IEC 61499 elements Composite Function Block and Subapplication. In *Proc. of INDIN'07*, Vol. 2. 1169–1175. https://doi.org/10.1109/INDIN.2007.4384941

[57] Thomas Vogel and Holger Giese. 2014. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8, 4 (2014), 18:1–18:33. https://doi.org/10.1145/2555612

[58] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. 2015. Evolution of software in automated production systems: Challenges and research directions. *J. Syst. Softw.* 110 (2015), 54–84. https://doi.org/10.1016/j.jss.2015.08.026

[59] Valeriy Vyatkin. 2010. The IEC 61499 standard and its semantics. *Industrial Electronics Magazine, IEEE* 3 (01 2010), 40 – 48. https://doi.org/10.1109/MIE.2009.934796

[60] Valeriy Vyatkin. 2011. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *Ind. Informatics, IEEE Transactions* 7 (2011), 768 – 781. https://doi.org/10.1109/TII.2011.2166785

[61] V. Vyatkin and H.-M. Hanisch. 1999. A modeling approach for verification of IEC1499 function blocks using net condition/event systems. In *Proc. of ETFA'99*, Vol. 1. 261–270 vol.1. https://doi.org/10.1109/ETFA.1999.815365

[62] Danny Weyns, Radu Calinescu, Raffaela Mirandola, Kenji Tei, Maribel Acosta, Amel Bennaceur, Nicolas Boltz, Tomás Bures, Javier Cámara, Ada Diaconescu, Gregor Engels, Simos Gerasimou, Ilias Gerostathopoulos, Sinem Getir Yaman, Vincenzo Grassi, Sebastian Hahner, Emmanuel Letier, Marin Litoiu, Lina Marsso, Angelika Musil, Juergen Musil, Genaína Nunes Rodrigues, Diego Perez-Palacin, Federico Quin, Patrizia Scandurra, Antonio Vallecillo, and Andrea Zisman. 2023. Towards a Research Agenda for Understanding and Managing Uncertainty in Self-Adaptive Systems.

ACM SIGSOFT Softw. Eng. Notes 48, 4 (2023), 20–36. https://doi.org/10.1145/3617946.3617951

[63] Danny Weyns, Ilias Gerostathopoulos, Nadeem Abbas, Jesper Andersson, Stefan Biffl, Premek Brada, Tomas Bures, Amleto Di Salle, Matthias Galster, Patricia Lago, Grace Lewis, Marin Litoiu, Angelika Musil, Juergen Musil, Panos Patros, and Patrizio Pelliccione. 2023. Self-Adaptation in Industry: A Survey. *ACM Trans. Auton. Adapt. Syst.* (mar 2023). https://doi.org/10.1145/3589227 Just Accepted.

[64] Danny Weyns and M. Usman Iftikhar. 2023. ActivFORMS: A Formally Founded Model-based Approach to Engineer Self-adaptive Systems. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 12:1–12:48. https://doi.org/10.1145/3522585

[65] Bianca Wiesmayr, Lisa Sonnleithner, and Alois Zoitl. 2020. Structuring Distributed Control Applications for Adaptability. In *Proc. of ICPS'20*, Vol. 1. 236–241. https://doi.org/10.1109/ICPS48405.2020.9274744

[66] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. 2009. Formal methods: Practice and experience. *ACM Comput. Surv.* 41, 4 (2009), 19:1–19:36. https://doi.org/10.1145/1592434.1592436

[67] Li Hsien Yoong, Partha S. Roop, Zeeshan Ejaz Bhatti, and Matthew M. Y. Kuo. 2015. *Model-Driven Design Using IEC 61499 - A Synchronous Approach for Embedded and Automation Systems.* Springer. https://doi.org/10.1007/978-3-319-10521-5

[68] Alois Zoitl and Robert Lewis. 2014. *Modelling control systems using IEC 61499. 2nd Edition.* Institution of Engineering and Technology. https://doi.org/10.1049/PBCE095E

[69] Alois Zoitl and Herbert Prähofer. 2013. Guidelines and Patterns for Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language. *IEEE Trans. Ind. Informatics* 9, 4 (2013), 2387–2396. https://doi.org/10.1109/TII.2012.2235449

[70] Vladimir Zyubin, Igor S. Anureev, Natalya Olegovna Garanina, Sergey M. Staroletov, Andrei Rozov, and Tatiana V. Liakh. 2021. Event-Driven Temporal Logic Pattern for Control Software Requirements Specification. In *Proc. of FSEN'21 (LNCS, Vol. 12818)*. Springer, 92–107. https://doi.org/10.1007/978-3-030-89247-0_7