Runtime Enforcement for IEC 61499 Applications

Yliès Falcone, Gwen Salaün, Irman Faqrizal

Runtime Enforcement for IEC 61499 Applications

Yliès Falcone, Irman Faqrizal, Gwen Salaün

Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

Abstract. Industrial automation is a complex process involving various stakeholders. The international standard IEC 61499 helps to specify distributed automation using a generic architectural model, targeting the technical development of the automation. However, analysing the correctness of IEC 61499 models remains a challenge because of their informal semantics and distributed logic. We propose new verification techniques for IEC 61499 applications. These techniques rely on the concept of runtime enforcement, which can be applied to systems for preventing *bad* behaviours from happening. The main idea of our approach is to integrate an enforcer in the application for allowing it to respect specific properties when executing. The techniques begin with the definition of a property. The language of this property supports features such as discarding and replacing events. Next, this property is used to synthesise an enforcer in the form of a function block. Finally, the synthesised enforcer is integrated into the application. Our approach is illustrated on a realistic example and fully automated.

1 Introduction

The emerging industrial revolution, Industry 4.0, affirms that the innovation of technologies has become the main driving force in the advancement of industrial activities [15]. Stakeholders in the industry appeal for new technologies in every aspect of industrial processes. These include the improvements of development tools for industrial automation to increase efficiency and productivity. The International Electrotechnical Commission (IEC) 61499 [1] is a recent standard for developing industrial automation. It conceptualises interconnected function blocks to express an industrial process. Each Function Block (FB) encapsulates some logic describing its behaviour, while the connections with other FBs are defined using input and output interfaces.

The main benefit of IEC 61499 is that it is suitable for developing a fully distributed system [19]. A single application can be distributed among several control devices to optimise efficiency. However, this advantage also raises a challenge because when the system is complex and composed of many control devices and FBs, it becomes error-prone. This a critical issue since IEC 61499 does not define how to handle bugs (e.g., there is no exception handling). Furthermore, the execution of industrial systems is heavily influenced by the environment. In contrast to conventional programs with mostly user interactions, industrial

applications also accept inputs from the connected sensors. In complex systems, there can be many sensors, and each of them is associated with the outside world, which has unpredictable behaviour.

IEC 61499 applications can be verified during design time using static verification techniques such as model checking [3, 14, 20].

Such an approach is useful for finding unexpected behaviour before the application is deployed. However, industrial applications can be huge, and state space explosion may become an issue. On top of that, the debugging process might introduce new bugs since it is done manually by the users. Also, IEC 61499 has loosely defined semantics, which causes the faithfulness of its translation from an application into a model can not be guaranteed (i.e., the model may not really represent the actual behaviour of the application). Furthermore, as previously mentioned, industrial applications often interact with nondeterministic behaviours of the environments, which can not be observed during design time. The work in [5] and [12] propose alternative methods to verify IEC 61499 applications by applying runtime verification techniques [7]. The main idea of both works is to integrate a monitor that can check during runtime whether some properties hold. These techniques can be applied regardless of the application's size, and there is no modelling phase required, which means that it is not necessary to define the application's formal semantics. In addition, the approach involves analyses of execution traces that are obtained directly from executing the application (taking into account the influence of environments). However, the users are still required to manually debug the application when the properties are violated. Moreover, in this case, the properties' violations can be detected only when the application is already running. This is an issue since applications with incorrect behaviours may cause a critical impact on industrial activities.

A practical solution for supporting IEC 61499 applications is to integrate verification techniques that can ensure correctness during runtime. To do so, we propose to rely on runtime enforcement [8] techniques for preventing systems from producing incorrect behaviours. It guarantees correctness by modifying the system execution. There exist multiple enforcement mechanisms such as *input sanitation*, which ensures correctness by altering inputs that enter the system, and *output sanitation*, which modifies outputs such that they follow certain requirements. In our case, we enforce the system's correctness by altering the outputs of existing components (i.e., FBs) in the application.

The main idea of our runtime enforcement techniques is to change the execution of IEC 61499 applications such that they can respect some given properties. For this purpose, the application is modified by integrating a new component called *enforcer* in the form of an FB. This enforcer is synthesised from an input property. Its purpose is to instrument the modification of the application's execution. Therefore, the property does not only specify the correctness of an application but also describes how the enforcement mechanism can react when the property violation happens. To achieve this, we define properties as automata extended with different types of transitions. A transition in the property can either let outputs be forwarded to the next component, discarded, or replaced

with modified outputs. Since an enforcer is a type of basic FB, the synthesis process includes the creation of FB interfaces, Execution Control Chart (ECC), and algorithms. Every element in this FB is derived from the input property. After an enforcer is synthesised, then it should be integrated into the application by appropriately connecting the input and output interfaces without changing the initial execution flow.

More precisely, our contributions are as follows: (i) a property language for enforcing IEC 61499 applications, (ii) a technique for synthesising enforcers (in the form of IEC 61499 FBs) from given properties, (iii) a sequence to integrate enforcers into IEC 61499 applications. The approach is illustrated on a realistic example, and tool support was developed to automate the synthesis of enforcers.

The paper is organised as follows. Section 2 introduces background notions. Section 3 describes the runtime enforcement techniques. Section 4 presents the supporting tools. Section 5 surveys related work and Section 6 concludes.

2 Background

We first present the necessary concepts from IEC 61499 (Section 2.1), followed by a description of the essential idea of runtime enforcement. In the next section, we show a running example to illustrate our approach.

2.1 IEC 61499

IEC 61499 is a standard for designing industrial control systems that consist of interconnected Function Blocks (FBs). A FB is connected to other FBs through its input and output interfaces, where each of them distinguishes between event and data interfaces (see Figure 1 (a)). The standard adopts an event-driven architecture, meaning that the execution of each component (i.e., FB) is triggered by incoming events. Once the FB is activated by an event, it cannot be re-entered by another event before the previous activation has finished. The *WITH* identifiers associate event and data interfaces. When an event arrives, the values on the associated input data interfaces are refreshed, whereas the emission of an event refreshes the associated values on the output data interfaces.

Figure 1 (b) illustrates IEC 61499. The standard allows the event interfaces to be connected in both fan-in and fan-out configurations (e.g., FB 1 to FB 2 and 3, while FB 2 and 3 to FB 4). However, an input data interface can only receive a connection from a single output data interface (i.e., fan-out only). Moreover, in the same picture, all the FBs (FB 1 to 5) are part of a single application (Application 1). However, as we can see, some of the FBs are mapped into different control devices. This emphasises the fact that IEC 61499 allows for designing distributed applications.

Function Block is the fundamental component of IEC 61499 architecture. There are three types of FBs: basic, composite, and service interfaces. A basic FB defines its behaviour using a state machine called Execution Control Chart (ECC). When a state is visited, it may perform two actions: emit an output

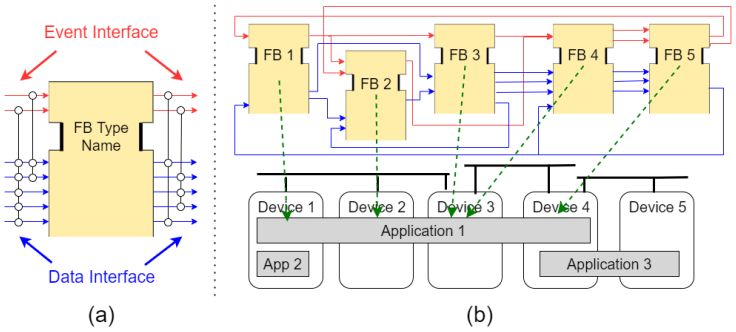

Fig. 1: Example of (a) Function block and (b) IEC 61499 application.

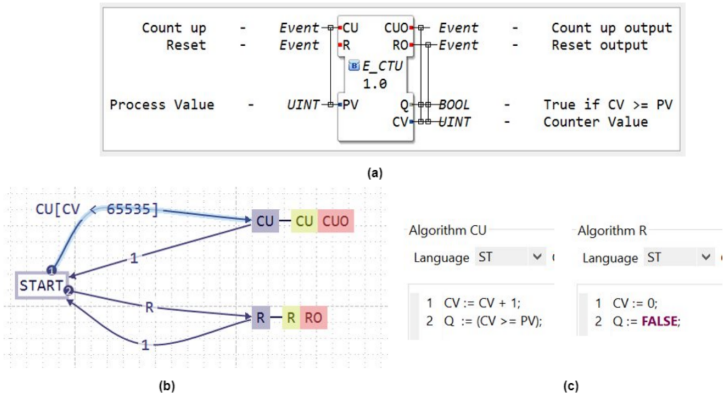

Fig. 2: (a) E_CTU FB, (b) ECC, (c) Algorithms.

event and/or execute an algorithm written in structured text. Some conditions on the data interfaces can guard the transitions in an ECC. A composite FB is composed of a network of FBs. A service interface FB concerns FBs that have behaviours specific to their control devices (i.e., vendor dependent).

In our work, we use 4DIAC-IDE [2] as a development environment, an open source tool to build IEC 61499 applications. Figure 2 (a) shows an example of FB represented with 4DIAC-IDE. This FB is called *E_CTU* and behaves as a counter. The ECC in Figure 2 (b) describes the FB behaviour in each activation. Starting from *START* state, if the FB receives an event *CU* and the value of *CV* is below the threshold (65535), then the current state transitions into *CU*. Such transition with a boolean condition is called guarded transition. Next, in state *CU*, algorithm *CU* (Figure 2 (c)) is executed and output event *CUO* is fired. The algorithm increments *CV* and updates the boolean value *Q*. Finally, it goes back directly to the initial state because the transition going from state *CU* to *START* is an empty transition. The same mechanism also applies when the FB receives event *R*, which resets the counter value *CV*.

2.2 Runtime Enforcement

Runtime enforcement [17] (see [6, 8, 10] for overviews) is a technique that can prevent systems from *misbehaving* by forcing them to execute according to their specifications. A specification is often formalised as properties to be satisfied.

The main goal of runtime enforcement techniques is to define the relation between input and output sequences of events. More precisely, the techniques describe how an incorrect sequence can be modified into a correct one. For this, a so-called *Enforcement Mechanism* (EM) transforms an input σ into an output $EM(\sigma)$ according to a property φ. There are three ways of implementing an EM. According to the terminology in [8], our EM is an *output sanitiser* since it prevents a system from generating incorrect traces.

There exist several (mathematical) models of EM. A *Security Automaton* [17] (SA) is a finite-state machine executing in parallel with the monitored program. When the model observes an action, the enforcer can either let it execute or halt the system. An extension of SA is the *edit-automata* [13] (EA), it has a feature that can suppress, memorise and replay actions. *Generalised enforcement monitors* [9] (GEMs) go further by separating sequence recognition from action memorisation. This simplifies the implementation and composition of operations.

2.3 Running Example

A running example is used in the rest of the paper to illustrate the runtime enforcement approach. The example is a conveyor test station, one of the case studies of IEC 61499 applications introduced in [21].

Figure 3 presents the running example. Its goal is to check the quality of industrial materials passing through a conveyor belt. The application consists of four main components. Firstly, a conveyor drive (C1) is connected to a control panel where the user can either start or stop the conveyor. Secondly, the component feeder (C2) is in charge of feeding materials onto the conveyor. Next, a quality acceptance station (C3) evaluates the materials as they pass through. Lastly, depending on the test results, the roll-off mechanism (C4) allows the materials either to be distributed onto the next industrial process or to be dropped into a hopper by opening the reject gate.

The IEC 61499 application of the conveyor test station is presented in Figure 4. Each of the main components is mapped to a composite FB. *DriveCntl1* corresponds to the conveyor drive (C1), *Feed1* represents the material feeder (C2), *QualStation1* deals with the quality acceptance station (C3), and *RollOff1* takes care of the roll-off mechanism (C4). The two additional FBs, *Exec1* and *Inventory1*, respectively correspond to the initialisation of the application and the database which stores the materials data. In these FBs, there are event and data interfaces that correspond to the application's functionalities. For instance, *Running* data interface in *DriveCntl1* has a boolean value which represents the state of the conveyor belt. When the conveyor is running then $Running = true$, otherwise $Running = false$. Inside each of these composite FBs, service interface FBs interact with the physical sensors and actuators such as *IO_READER* and

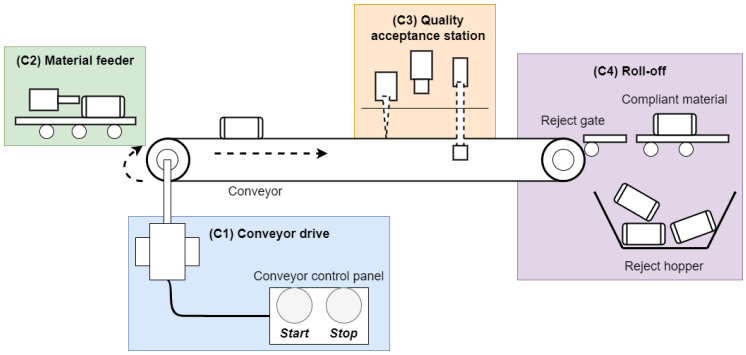

Fig. 3: Conveyor test station

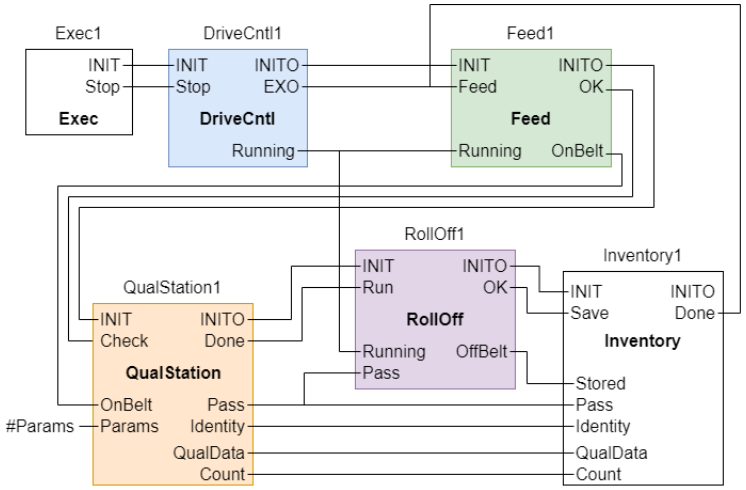

Fig. 4: IEC 61499 application of the conveyor test station

IO_WRITER. For example, an *IO_READER* FB in *Feed1* interacts with the input sensor to detect that a piece of material is successfully fed onto the conveyor. For brevity, we leave out the details of what is internal of every composite FBs; a more comprehensive description can be found in [21].

3 Runtime Enforcement Techniques

This section describes the runtime enforcement techniques for IEC 61499 applications. First, the enforcement architecture is presented, and the property language is explained. We then describe the synthesis of an enforcer and how to integrate it into the application. The section ends with a description of preserved characteristics.

3.1 Enforcement Architecture

Figure 5 illustrates the general architecture of our enforcement techniques for IEC 61499 applications. It is composed of a monitored application and an enforcer synthesised from a property. Monitored components are certain FBs in the application for which we want to ensure their correctness based on specific properties, Whenever one of these FBs outputs an event e with its associated data updates D, the enforcer intercepts this output and alters it according to the specified property. The altered outputs (e', D') are then forwarded to the next connected FBs in the application. The enforcer is synthesised as an FB. Therefore, it also has input and output interfaces for receiving and triggering events with the associated data updates. Its ECC and algorithms compute the output every time an input is received. Lastly, an enforcer has to be integrated into the application by creating new connections between the interfaces of the enforcer and the monitored components.

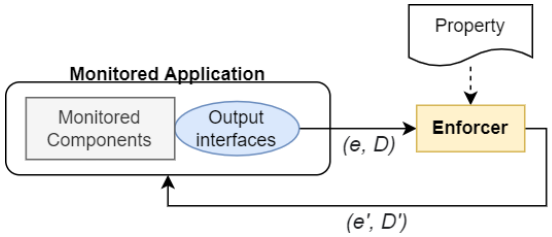

Fig. 5: Overview of the enforcement architecture

3.2 Property Automaton

A property is required as an input of the runtime enforcement techniques. We express properties as automata. This allows enforcing properties in any alternative declarative logical formalisms (e.g., Linear Temporal Logic) that translate to automata. Each automaton consists of states interconnected with transitions. A state can either be a correct state or an incorrect state, which we denote using the colours green and red. A property is satisfied when the current state is green and violated when red. A transition outgoing to a red state corresponds to the property violation itself.

Transitions are extended with types that are used to describe the behaviour of the synthesised enforcer every time it receives an input. There are four types of transitions:

- A *forward* transition indicates that the triggered event and its associated data updates are forwarded (i.e., nothing changed).
- A *discard* transition indicates that the event is discarded and no data is updated.
- A pair of transitions *replace* and *replacement* outgoing from the same state indicates that the triggering of an event and the updating of data interfaces

on transition typed as *replace* should be replaced with the triggering of an event and the updating of data interfaces on another transition typed as *replacement*.

A transition typed as *forward* or *replacement* is always outgoing from a green state to another green state. Meanwhile, *discard* and *replace* transitions are outgoing from green states to red states.

Definition 1. *(IEC 61499 property) A property for IEC 61499 enforcement mechanism is an automaton* $P = (S, s^0, E, B, \Gamma, T, va)$*, where*

- S *is a (finite) set of states, and* s^0 *is the initial state,*
- E *is a set of events,*
- B *is a set of boolean expressions,*
- $\Gamma = \{forward, discard, replace, replacement\}$ *is a set of transition types,*
- T *is the set of transitions and each* $t \in T$ *is a transition* $t = (s, e, G, \gamma, s')$*, where* $s, s' \in S$ *are source and target states,* $e \in E$ *is an event,* G *is a boolean guard composed of* $b \in B$ *and generated by the grammar* $G ::= true \mid b \mid \neg b \mid G \wedge G \mid G \vee G$*, and* $\gamma \in \Gamma$ *is the transition's type,*
- $va : S \to \{green, red\}$ *is the verdict function associating each state to a colour.*

Boolean expressions existing in a transition's guard represent the values of data interfaces when the associated event is triggered. The expression *true* implies that the event on that transition is triggered regardless of the data interfaces' current values. When a transition is typed as *replacement*, the guard must refer to a single possible combination of values, i.e., every boolean expression uses only equality operator ($=$) and only conjunction ($\wedge$) is allowed before or after each expression. The purpose is to ensure a unique replacement when performing data updates on transitions typed as *replace*.

Two examples of properties are shown in Figure 6. Both of them are specified for IEC 61499 application introduced in Section 2.3. We call the first property as Regulate Buttons. It is associated with conveyor drive component (C1) or *DriveCntl1* FB. It specifies that every time event *EXO* is triggered, the value of *Running* can only alternate between *true* and *false*. This is done by discarding event*EXO* guarded with $Running = false$ outgoing from state 0 and *EXO* guarded with $Running = true$ outgoing from state 1. In practice, this property helps to suppress the impact of users consecutively pressing the same button on the control panel.

The property in Figure 6 (B) is called Force Accept. It enforces the behaviour of the quality acceptance station component (C3) or *QualStation1* FB. It permits the application to reject industrial materials only twice in a row, and the third rejection is forced to be an acceptance instead. This is done by specifying that the event *Done* guarded by $Pass = false$ (i.e, material is rejected) can only be triggered twice in a row (i.e., transitions $Done.\{Pass = false\}.(forward)$ from state 1 to 2 and from state 2 to 3). The third time it is triggered (from state 3 to -1), then it is replaced with the transition $Done.\{Pass = true\}.(replacement)$ from state 3 to 1, where $Pass = true$ means that a material is accepted.

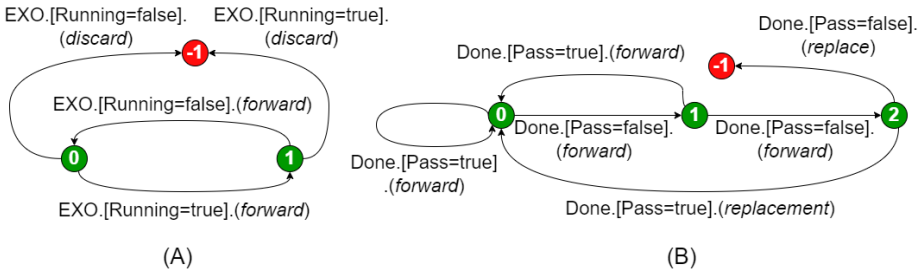

Fig. 6: Examples of properties, (A) Regulate Buttons and (B) Force Accept

3.3 Enforcer Synthesis

An enforcer is an FB synthesised from a given property. It is in the form of a basic FB, and it has interfaces, ECC, and algorithms. The idea is to integrate the enforcer as an additional FB in the IEC 61499 application to enforce its behaviour according to the property. The components of an enforcer are derived from the property.

Definition 2. *(Enforcer) An enforcer is a basic FB* $ef = (ei, ecc, A)$*, where:*

- $ei = (E_i, E_o, D_i, D_o, W_i, W_o)$ *is an enforcer interface, where* E_i, E_o *are sets of input and output event interfaces,* D_i, D_o *are sets of input and output data interfaces,* W_i *and* W_o *are WITH input and output identifiers associating events and sets of data interfaces,*
- $ecc = (S_c, s_c^0, B_c, T_c)$ *is an Execution Control Chart (ECC) specifying the enforcer's behaviour, where* S_c *is a set of states and* $s_c^0 \in S_c$ *is the initial state and each* $s_c \in S_c$ *consists of state actions* $s_c = q_1, q_2, ..., q_n$*, each action is a pair* $q = (a, e_o)$*, where* $a \in A$ *is an algorithm and* e_o *is an output event,* B_c *is a set of boolean expressions on* D_i*,* T_c *is the set of ECC transitions, each* $t_c \in T_c$ *is a transition* $t_c = (s_c, e_i, G_c, s_c')$*, where* $s_c, s_c' \in S_c$ *are source and target states,* $e_i \in E_i$ *is an input event,* G_c *is a boolean guard composed of* $b_c \in B_c$ *and generated by the grammar* $G_c ::= true \mid b_c \mid \neg b_c \mid G_c \wedge G_c \mid G_c \vee G_c$*,*
- A *is a set of algorithms, where* $a \in A$ *consists of assignments for variables in* D_o*.*

The enforcer interface serves as a connection for integrating the enforcer into the application. It is synthesised based on the events and boolean expressions present in the property. For each event or expression variable in the property, a pair of input and output interfaces are created in the enforcer. The *WITH* identifiers are created from the associated events and expressions in guards. ECC is built by traversing the states and transitions in the property. It specifies the behaviour of an enforcer every time it receives an input event from one of the monitored components. Algorithms reside in the ECC's states; they contain value assignments of the data interfaces.

Algorithm 1 describes the synthesis of an enforcer. It takes as input a property P and returns an enforcer ef. Interfaces E_i, E_o, D_i, D_o are created by

Algorithm 1: Synthesis of enforcer

Inputs : $P = (S, s^0, E, B, \Gamma, T)$
Output: $ef = (ei, ecc, A)$

$E_i := \{e_i \mid e \in E\}$
$E_o := \{e_o \mid e \in E\}$ /* getVars() returns a set of variable */
$D_i := \{d_i \mid d \in getVars(B)\}$ /* names from a set of boolean expression */
$D_o := \{d_o \mid d \in getVars(B)\}$ /* or a Guard */
foreach $(s, e, G, \gamma, s') \in T$ **do**
  $D'_i := \{d'_i \mid d \in getVars(G)\}$
  $D'_o := \{d'_o \mid d \in getVars(G)\}$
  $W_i := W_i \cup \{(e + "\_I",\ D'_i)\}$
  $W_o := W_o \cup \{(e + "\_O",\ D'_o)\}$
end
$e_i = (E_i, E_o, D_i, D_o, W_i, W_o)$
$Visited := \varnothing,\ s^0_c := (\varnothing, \varnothing)$
Function `TraverseProperty`$(s, s_c, P, ecc, A, Visited)$**:**
  $Visited := Visited \cup \{s\}$
  let $T' \subseteq T$ be the set of transitions outgoing from state s **in**
    foreach $(s, e, G, \gamma, s') \in T'$ **do**
      foreach $b \in getExpressions(G)$ **do**
        $a := a\ +\ getVars(b)\ +\ ''\_O :=''\ +\ getVars(b)\ +\ ''\_I;''$
      $A := A \cup \{a\}$ /* getExpressions() returns a set of boolean */
      $e_i := e\ +\ ''\_I''$ /* expressions from a Guard, whereas, getVal() */
      $e_o := e\ +\ ''\_O''$ /* returns a value of data in an expression */
      $s'_c := (a, e_o)$
      if $\gamma \neq discard\ \&\ \gamma \neq replace$ **then**
        $T_c := T_c \cup \{(s_c, e_i, G, s'_c)\}$
      if $\gamma = replace$ **then**
        $(s_r, e_r, G_r, \gamma_r, s'_r) := t \in T'$ where $\gamma_r = replacement$
        foreach $b_r \in getExpressions(G_r)$ **do**
          $a_r := a_r\ +\ getVars(b_r)\ +\ '' :=''\ +\ getVal(b_r)\ +\ '';''$
        $s''_c := (a_r, e_r)$
        $T_c := T_c \cup \{(s_c, e_i, G, s''_c)\}$
      $S_c := S_c \cup \{s_c, s'_c, s''_c\}$
      if $s' \notin Visited$ **then**
        TraverseProperty $(s', s'_c, P, ecc, A, Visited)$
    end
End Function
TraverseProperty $(s^0, s^0_c, P, ecc, A, Visited)$
return $ef = (ei, ecc, A)$

concatenating property's events or variable names with the corresponding suffixes (lines 1 to 4). The suffixes are added since every interface in an FB must have a unique name. The *WITH* identifiers W_i, W_o are obtained by iterating through transitions in the property (lines 5 to 9) and associating each event with the set of data interfaces taken from variable names on the guard.

ECC is built by traversing the property using a recursive function *Traverse-Property* to visit every state. In each recursion:

1. an ECC's algorithm is written by obtaining every variable in the transition's guard and creating an assignment of input data interfaces to output data interfaces,
2. an ECC's transition is created and added into the set only when the type of property's transition is not *discard* (lines 21 and 25),
3. an additional ECC's transition is added when there are a pair of *replace* and *replacement* transitions (lines 26 to 30),
4. the set of ECC's states is updated and proceeds to the next state when it is not yet visited (lines 31 to 33).

Table 1: Synthesised Enforcers Interfaces

Interface	**Regulate Buttons**	**Force Accept**
Input event	*EXO_I*	*Done_I*
Output event	*EXO_O*	*Done_O*
Input data	*Running_I*	*Pass_I*
Output data	*Running_O*	*Pass_O*
WITH input	$(EXO\_I, \{Running\_I\})$	$(Done\_I, \{Pass\_I\})$
WITH output	$(EXO\_O, \{Running\_O\})$	$(Done\_O, \{Pass\_O\})$

As examples of synthesis results, we first present synthesised enforcers interfaces in Table 1. These are enforcer interfaces generated from the properties presented in Figure 6 (A) and (B). Enforcer interfaces are generated from the property Regulate Buttons in the second column, whereas enforcer interfaces are generated from the property Force Accept in the third column. For each property, there is a pair of event input and output interfaces (e.g., *EXO_I* and *EXO_O*) generated from an event. There is also a pair of data input and output (e.g., *Running_I* and *Running_O*) since there exists only one variable in the boolean expression. The *WITH* identifiers associate each event interface with a set of data interfaces according to events and guards in the properties' transitions. For instance, in Regulate Buttons *EXO_I* is associated with *Running_I* because, in property, there is a transition with event *EXO* associated with a guard that contains data variable *Running*.

Figure 7 depicts the synthesised ECCs of enforcers from both properties, and Table 2 shows the algorithms. The Regulate Buttons property contains *discard*

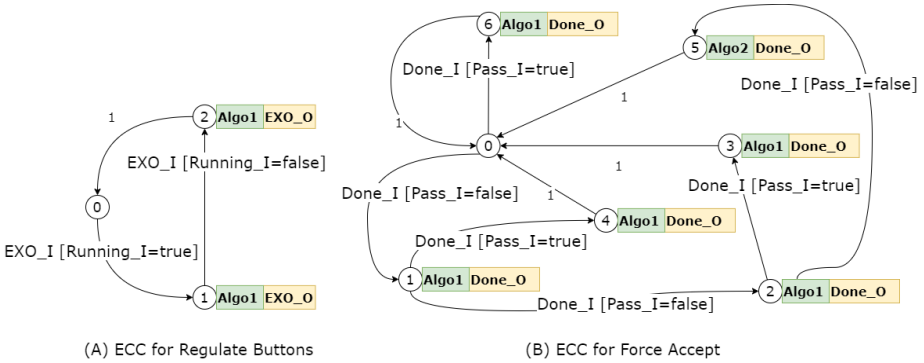

Fig. 7: Synthesised Execution Control Charts

transitions. It translates into an ECC where the corresponding transitions are removed. For instance, in state 1 when the enforcer has just triggered *EXO* and set *Running* to *true*, there is only a single transition where it can receive *EXO* with *Running* set to *false*. When the enforcer receives an event *EXO* with *Running* = *true*, then that event is discarded, and the current state stays at state 1. Hence, the value of *Running* always alternates between *true* and *false* every time *EXO* is triggered. Meanwhile, in the enforcer ECC for the Force Accept property, state 2 corresponds to state 3 of property. In this state, the enforcer has received *Done* with *Pass* set to *false* (i.e., material rejection) twice in a row. Notice that from this state when the enforcer receives event *Done* with *Pass* set to *false* for the third time, state 5 executes *Algo2* where the value of *Pass* is enforced to be *true*. Furthermore, when the property's transition is typed as *forward*, the generated ECC's target state simply uses an algorithm where we assign the value of the input data interface to the output data interface. As an example for the property Force Accept, *Algo1* which assigns *Pass_I* to *Pass_O* is used in every state except state 5.

Table 2: Synthesised Algorithms

	Regulate Buttons	**Force Accept**
Algo1	`Running_O := Running_I;`	`Pass_O := Pass_I;`
Algo2		`Pass_O := true;`

3.4 Enforcer Integration

A synthesised enforcer must be integrated into an IEC 61499 application in order to enforce the property's correctness at runtime. Enforcer integration is illustrated in Figure 8; below is the description of every step:

(1) Identify the subset of output interfaces in the application by matching their names with enforcer input interfaces (e.g., *EO* and *DO*).
(2) Identify the subsets of input interfaces connected with output interfaces identified in step 1 (e.g., $\{EI1, EI2\}$ and $\{D\}$).
(3) Connect the output interfaces identified in step 1 to the input interfaces of the enforcer (e.g., *EO* to *EO_I*) and connect the output interfaces of the enforcer with the input interfaces identified in step 2 (e.g., *EO_O* to *EI1* and *EI2*).
(4) Disconnect output interfaces in step 1 and input interfaces in step 2.

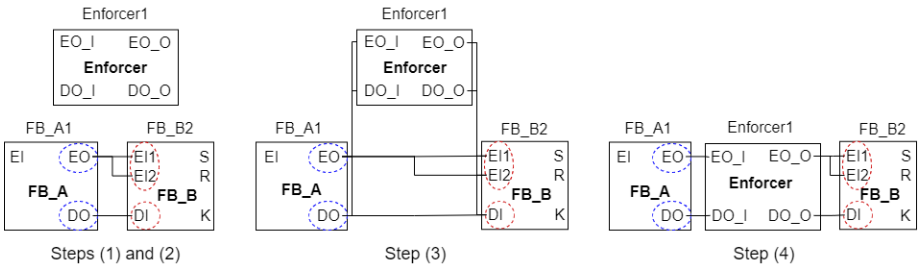

Fig. 8: Four Steps of Enforcer Integration

Figure 9 shows excerpts of enforcers for properties in Figure 6 that are integrated into the application. Integrating an enforcer essentially places a new FB between sets of connections. For instance, output event interface *EXO* in *DriveCntl1* was initially connected to *Feed* in *Feed1*. After the enforcer is integrated, this connection is replaced with *EXO* to *EXO_I* and *EXO_O* to *Feed*.

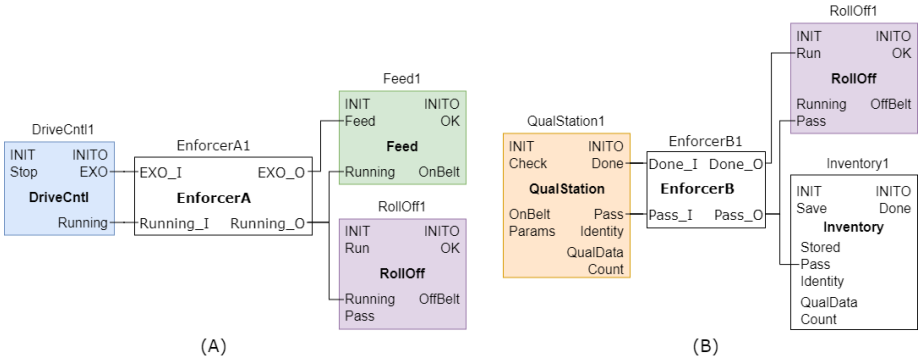

Fig. 9: Integrated Enforcers

3.5 Characteristics

Our approach involves the modification of the application. It is essential to make sure that this modification respects some common characteristics. The first characteristic, *soundness* is satisfied if, for any input, it produces the correct output

which satisfies the property. This criterion is fulfilled because we synthesise enforcers directly from properties. In a property, transitions outgoing to an incorrect state are either typed as *discard* or *replace*. Hence, any incorrect input is always either discarded or replaced. The second characteristic, *transparency*, is satisfied if the enforcer only intervenes when a property violation happens. This is also true in our approach since the enforcer only replaces events and data when there is a transition typed as *replace* outgoing to an incorrect state (i.e., property's violation). Also, the enforcer only discards an event when there is a transition typed as *discard* outgoing to an incorrect state.

4 Tool Support

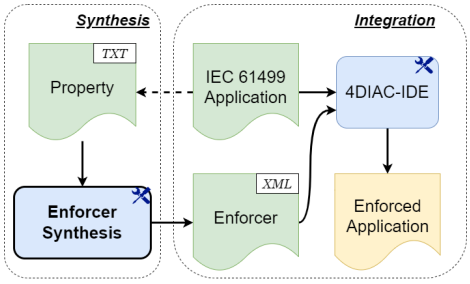

Fig. 10: Implementation Overview

We have developed tool support in Java programming language to synthesise enforcers automatically. The tool takes as input a property and outputs an enforcer. The resulting enforcers can be visualised and simulated using Eclipse 4DIAC-IDE [18]. Figure 10 overviews our implementation. A property is written in a text file for a certain IEC 61499 application. The enforcer synthesis takes this property as an input to generate an enforcer in XML format. 4DIAC-IDE is then used to integrate the enforcer into the application. The application with an integrated enforcer is then ready to be simulated. The tool support, along with the running example in Section 2.3 integrated with enforcers for properties in Figure 6 are available online [11]. By taking advantage of this tool support, we have also done several experiments using other examples such as capping station [22] and temperature control [21], both with a variety of properties.

5 Related Work

Several formal verification techniques for IEC 61499 applications have been proposed [4]. The work in [14] introduces a technique based on model checking to visually explain properties' violations. The approach begins with automatic translations of IEC 61499 applications into Symbolic Model Verifier (SMV) specifications. Then, a model checker generates counterexamples from those specifications and some given properties. Finally, the counterexamples are utilized to

infer influence paths in a graphical interface. These paths are presented visually to the users to help them debug IEC 61499 applications. Earlier work in [20] uses Esterel for verifying safety properties. These works focus on avoiding properties from being violated by checking them before the application's deployment, whereas our approach ensures those properties are satisfied during runtime by applying modifications. Moreover, in this approach, the state explosion can be a challenge when the application is huge since it has to explore all possible executions. In contrast, our approach is not limited by the application's size.

A runtime verification technique is proposed in [12]. The authors propose a method for monitoring adapter connections in IEC 61499 application. The monitor contains state machines specifying certain properties. When a property is violated, an event is triggered as a notification. Instead of inserting a monitor, our work ensures correctness by integrating an enforcer based on a given property. The work in [5] also relies on runtime verification. The authors propose to integrate so-called *contract monitors* into IEC 61499 applications. These monitors can ensure some specified properties during runtime by constraining the behaviour of existing FBs. This is done by allowing data input and output interfaces to receive and send certain values only. However, this approach ensures only the correctness of individual FBs in the application, whereas our enforcer can ensure a property that involves multiple FBs at once.

The work in [16] introduces a technique to generate sequences of dynamic reconfiguration for IEC 61499 applications. The purpose of a reconfiguration sequence is to guarantee the continuity of a running application when component modifications are being applied. The technique takes as inputs an initial application and a target application. It outputs the sequence of reconfiguration steps to achieve the input target application from the initial input application. The technique starts by using predefined dependency rules on the input application to produce a dependency tree. This tree is then used to generate the sequence of reconfiguration. This approach does not guarantee that the target application is correct since it is an input the users give. In contrast, our approach involves automatically synthesising enforcers that guarantee correct behaviour when integrated into the application.

Compared to the existing models of enforcement monitors (EMs) [8], we consider output sanitisers that alter the outputs of existing components (i.e., FBs) in the application and forward them to the following connected components. Contrarily to the standard runtime enforcement scenario, EMs do not run in parallel with the application but are instead incorporated into the application. The EMs modify the application execution during runtime. Our EMs resemble edit automata [13] and generalised enforcement monitors [9] in that they can suppress and replace actions from the underlying application. However, they are synthesised from richer properties where events are not propositional (as with EAs and GEMs) but carry data values from the application that directly influence the decisions of our EMs.

6 Concluding Remarks

We propose new techniques to support industrial applications developed using the IEC 61499 standard. The novelty of this work is that we apply runtime enforcement to prevent IEC 61499 applications from violating certain properties during runtime. This approach allows the developers to guarantee correctness without manual intervention. Our enforcement techniques involve integrating a new component called an enforcer into the application. This component is in the form of a basic function block which is synthesised from a property. The approach is illustrated on a realistic running example, and tool support was developed to automate the synthesis process.

For future work, the property and the synthesised enforcer can be extended to support buffering. This feature would allow the enforcer to buffer events and trigger them in the future. This could be useful, for instance, when we need to postpone some actions due to a lack of resources in the system. With this feature, we could buffer the events corresponding to those actions and trigger them consecutively when the data corresponding to the resources are available. Another perspective is to construct an enforcer that can interpret properties dynamically. We may achieve this by designing enforcers as service interface FBs. With this type of FB, a program which implements a property interpreter can be written directly into the enforcer. Users could thus dynamically provide properties as inputs to change the enforcer behaviour at runtime. Furthermore, according to our preliminary experiments, the proposed enforcement techniques did not induce noticeable overheads. However, we plan to run more exhaustive analysis to verify that the approach does not significantly impact the application's performance.

Acknowledgements. 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. International Electrotechnical Commission, Functional blocks - Part 1: Architecture, 2nd edn, IEC 61499-1. *IEC Geneva*, 2012.
2. 4DIAC-IDE (2010). Framework for Distributed Industrial Automaton (4DIAC). `https://www.eclipse.org/4diac/`.
3. C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
4. J. O. Blech, P. Lindgren, D. Pereira, V. Vyatkin, and A. Zoitl. A comparison of formal verification approaches for IEC 61499. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2016.
5. D. Do Tran, J. Walter, K. Grüttner, and F. Oppenheimer. Towards time-sensitive behavioral contract monitors for IEC 61499 function blocks. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, volume 1, pages 27–34, 2020.
6. Y. Falcone. You should better enforce than verify. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St.*

Julians, Malta, November 1-4, 2010. Proceedings, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.
7. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 01 2013.
8. Y. Falcone, L. Mariani, A. Rollet, and S. Saha. *Runtime Failure Prevention and Reaction*, pages 103–134. Springer International Publishing, Cham, 2018.
9. Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38, 06 2011.
10. Y. Falcone and S. Pinisetty. On the runtime enforcement of timed properties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 48–69. Springer, 2019.
11. I. Faqrizal. Enforcer Synthesis. `https://gitlab.inria.fr/ifaqriza/enforcer-synthesis`, 2022.
12. P. Jhunjhunwala, J. O. Blech, A. Zoitl, U. D. Atmojo, and V. Vyatkin. A design pattern for monitoring adapter connections in IEC 61499. In *22nd IEEE International Conference on Industrial Technology, ICIT 2021, Valencia, Spain, March 10-12, 2021*, pages 967–972. IEEE, 2021.
13. J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In S. d. C. di Vimercati, P. Syverson, and D. Gollmann, editors, *Computer Security – ESORICS 2005*, pages 355–373, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
14. P. Ovsiannikova and V. Vyatkin. Towards user-friendly model checking of IEC 61499 systems with counterexample explanation. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 01–04, 2021.
15. T. Philbeck and N. Davis. The fourth industrial revolution: Shaping a new era. *Journal of International Affairs*, 72(1):17–22, 2018.
16. L. Prenzel and S. Steinhorst. Automated dependency resolution for dynamic reconfiguration of iec 61499. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2021.
17. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, feb 2000.
18. T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sünder, A. Valentini, and A. Martel. Framework for distributed industrial automation and control (4DIAC). *IEEE International Conference on Industrial Informatics (INDIN)*, pages 283 – 288, 08 2008.
19. V. Vyatkin. IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *Ind. Informatics, IEEE Transactions*, 7:768 – 781, 2011.
20. L. H. Yoong and P. S. Roop. Verifying IEC 61499 function blocks using esterel. *IEEE Embedded Systems Letters*, 2(1):1–4, 2010.
21. A. Zoitl and R. Lewis. *Modelling control systems using IEC 61499. 2nd Edition.* Institution of Engineering and Technology, 2014.
22. A. Zoitl, T. I. Strasser, and G. Ebenhofer. Developing modular reusable IEC 61499 control applications with 4DIAC. In *11th IEEE Int. Conf. on Ind. Informatics, INDIN*, pages 358–363. IEEE, 2013.