Guided Evolution of IEC 61499 Applications

Irman Faqrizal, Gwen Salaün, Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
Email: {irman.faqrizal, gwen.salaun}@inria.fr, ylies.falcone@univ-grenoble-alpes.fr

***Abstract*—IEC 61499 is a standard for developing industrial automation systems. It is known for its reusability, reconfigurability, interoperability, and portability. However, during their life cycle, industrial systems need to evolve according to requirements, and modifying the applications to satisfy these requirements can be complex and error-prone. This paper proposes techniques to guide the evolution of IEC 61499 applications. Given an initial application and the evolution requirements, we generate guidelines for modifying the application to satisfy the requirements. The application is first translated into a behavioural model describing all possible sequences of events the application can trigger. We then apply algorithms to extract relevant submodels of the application and modify them according to the requirements. Finally, the submodels are analysed to generate guidelines for modifying the application. These guidelines can bridge the gap between the requirements and the target application. Instead of only considering the requirements when exploring possible modifications, the developers can use the guidelines to make necessary changes to the application. A mixing tank system is used as a running example to illustrate the approach. In addition, a prototype to automate the evolution techniques is developed.**

***Index Terms*—Industrial automation systems, IEC 61499, software evolution**

I. Introduction

Software evolution is the process of modifying software due to various reasons, such as corrections, adaptations, improvements, and preventions [1]. As software evolves to satisfy the requirements [2], its size and complexity increase [3]. It is important to define a systematic software evolution method that guarantees the satisfaction of the requirements without introducing bugs or adding unnecessary complexity.

IEC 61499 [4] is a promising industrial automation standard. It supports reusability, reconfigurability, interoperability, and portability [5]. The standard defines the behaviour of a system using interconnected Function Blocks (FBs). During the life cycle of a system, this network of FBs may need to be modified to satisfy certain requirements. There are various ways to satisfy these requirements, such as modifying the FBs' internal functionalities and creating new connections between FBs. Manual modifications may result in unsatisfied requirements, unexpected behaviours, and extra complexity.

Most existing works, e.g., [6]–[11], focus on the dynamic reconfiguration of IEC 61499 applications. They propose methods to evolve an initial application into a target application without stopping the execution. In these works, there is no notion of requirement, and the target application is the user's input. To the best of our knowledge, techniques for computing the target application have not been proposed.

This work aims to guide the evolution of IEC 61499 applications according to requirements. Our approach is illustrated in Fig 1. It takes as input (1) an initial IEC 61499 application and (2) the evolution requirements. It generates as output (3) the guidelines to obtain (4) an evolved IEC 61499 application satisfying the requirements. We focus on applications that define the behaviours of industrial systems using interactions between the sensors and actuators [12]. Such applications are characterised by the presence of service interface FBs (SIFBs) that are associated with the sensors (i.e., sensor SIFBs) and the actuators (i.e., actuator SIFBs). Requirements are specified based on the expected interactions between the sensors and the actuators. The generated guidelines inform the developers about FBs and connections that need modification.

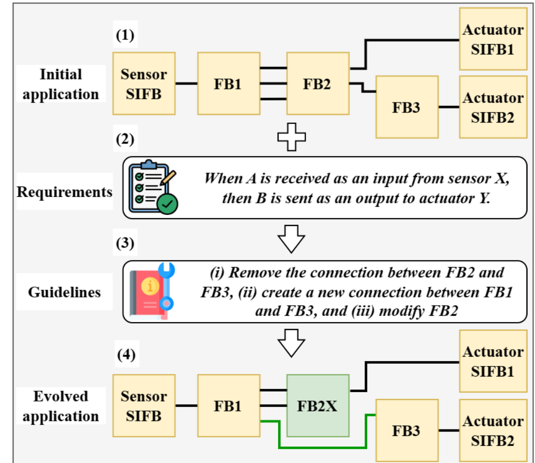


Fig. 1: Guided evolution of an IEC 61499 application

This paper's contribution consists of three modules. The first module translates the application into a behavioural model and interprets requirements as a specification. The second module identifies relevant submodels of the application and modifies them according to the specification. The last module analyses these submodels to generate the evolution guidelines. We apply our approach to a mixing tank system. In addition, a prototype to automate our method has been implemented. The method for generating application models in the first module is adapted from the work in [13]. This method has been applied to a manufacturing case study in [14].

This paper is structured as follows. Section II introduces background notions. Section III presents the running example. Section IV explains the guided evolution techniques. Section V describes the implementation of the approach. Section VI surveys related work. Section VII concludes.

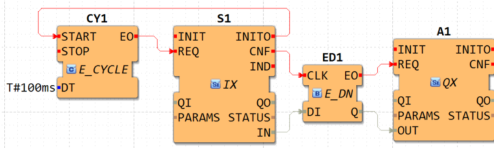


Fig. 2: Example of IEC 61499 application

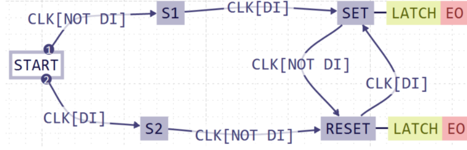


Fig. 3: The ECC of *ED1*

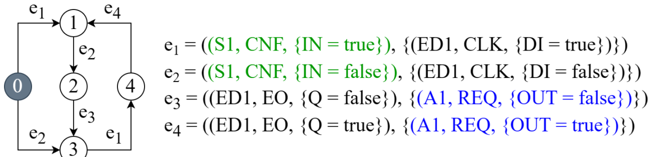


Fig. 4: Model of the IEC application in Fig. 2

II. Background

A. IEC 61499

IEC 61499 [4] is an international standard for developing industrial automation systems. It is the successor of IEC 61131-3 [15]. Unlike its predecessor, which uses a cyclic execution model, IEC 61499 adopts an event-driven execution model. The standard defines the notion of Function Block (FB). Each FB encapsulates an internal behaviour and can be connected to other FBs via its event and data interfaces. The internal behaviour of an FB is activated every time it receives an event. A set of interconnected FBs describes the behaviour of a system. IEC 61499 applications presented in this paper are designed using 4diac IDE [16].

There are three types of FB: basic, service interface, and composite. A basic FB encapsulates a state machine called the Execution Control Chart (ECC). A Service Interface FB (SIFB) provides access to the control device functionality, such as reading inputs from the sensors (i.e., sensor SIFB) or sending outputs to the actuators (i.e., actuator SIFB). A composite FB consists of a network of FBs.

An example of an IEC 61499 application is shown in Fig. 2. *CY1* is an FB that generates events periodically. *S1* and *A1* are sensor and actuator SIFBs. *ED1* is a basic FB with an ECC shown in Fig. 3. This ECC regulates the flow of events in the application to make the actuator SIFB only receive an event whenever the sensor SIFB's data value changes.

B. Model

In this work, the behaviour of an IEC 61499 application is modelled as a Labelled Transition System (LTS) [17]. A model consists of states and transitions labelled by events. An event is composed of an output and a set of inputs. The model of an application describes all possible sequences of events that the application can trigger. We rely on the translation patterns proposed in [13] to generate behavioural models from IEC 61499 applications. These patterns have been used to model real-world applications, such as a drilling station [14].

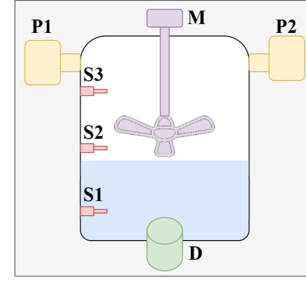


Fig. 5: Mixing tank system

Definition II.1 (Event)**.** *An event is a tuple* (o, I)*, where:*

- $o = (fb^o, eo, DO)$ *is an output consisting of a source FB* fb^o*, an event interface* eo*, and a set of data* DO*,*
- $I = \{(fb_1^i, ei_1, DI_1), (fb_2^i, ei_2, DI_2), ..., (fb_n^i, ei_n, DI_n)\}$ *is a set of inputs, each input consists of a target FB* fb^i*, an event interface* ei*, and a set of data* DI*.*

Definition II.2 (Model)**.** *The behaviour of an application is modelled as an LTS* (S, s_0, E, T)*, where:*

- S *is a set of states, and* $s_0 \in S$ *is the initial state,*
- E *is a set of events,*
- $T \subseteq S \times E \times S$ *is a set of transitions.*

Let us use the example in Fig. 2 to illustrate the notions of event and model. Suppose *S1* triggers an event from *CNF* with $IN = true$. Therefore, we can write this event as $((S1, CNF, \{IN = true\}), \{(ED1, CLK, \{DI = true\})\})$. Fig. 4 shows the model generated from the application in Fig. 2. The highlighted texts are the outputs (green) and inputs (blue) of the sensor and actuator SIFBs. The model shows the application's behaviour explained in Section II-A. For instance, in transition 2 to 3, the actuator SIFB *A1* receives an input on *REQ* with $OUT = false$ after the sensor SIFB *S1* sent two outputs sequentially from *CNF* with $IN = true$ (transition 0 to 1) then $IN = false$ (transition 1 to 2).

III. Running Example

A mixing tank system [18] in Fig. 5 is used in this paper as a running example to explain the guided evolution techniques. The system aims to mix two types of liquid coming from two different sources. S1, S2, and S3 are the sensors that detect the amount of liquid in the tank. P1, P2, M, and D are the actuators associated with two pumps, a mixer, and a drain.

The initial behaviour of the mixing tank is described in Table I (Behaviour 1). This behaviour is a standard mixing process in which both pumps start and stop pumping when the tank is empty and full. The liquid is drained when the tank is full, and the mixer is on when there is enough liquid. The other two behaviours in the table are the target behaviours of the

TABLE I: The initial and target behaviours of the mixing tank

Behaviour 1 (initial behaviour)	
1) When the liquid moves below S1, (i) P1 and P2 start pumping, and (ii) D stops draining. 2) When the liquid moves above S2, M starts mixing. 3) When the liquid moves above S3, (i) P1 and P2 stop pumping, and (ii) D starts draining. 4) When the liquid moves below S2, M stops mixing.	
Behaviour A	**Behaviour B**
1) **When the liquid moves below S1, (i) P1 starts pumping, and (ii) D stops draining.** 2) **When the liquid moves above S2, (i) P2 starts pumping, and (ii) M starts mixing.** 3) When the liquid moves above S3, (i) P1 and P2 stop pumping, and (ii) D starts draining. 4) When the liquid moves below S2, M stops mixing.	1) When the liquid moves below S1, (i) P1 and P2 start pumping, and (ii) D stops draining. 2) **When the liquid moves above S3, (i) P1 and P2 stop pumping, (ii) D starts draining, and (iii) M starts mixing.** 3) When the liquid moves below S2, M stops mixing.

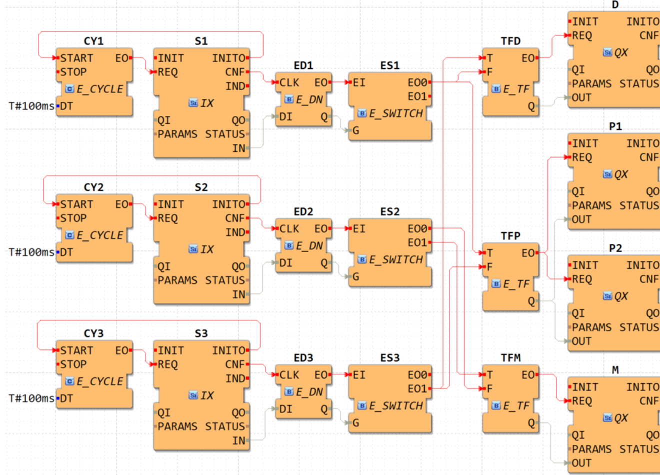


Fig. 6: Initial IEC 61499 application of the mixing tank system

evolved applications. The bold texts highlight the differences between the target and initial behaviours. In Behaviour A, we assume that the liquid supply from P2 is almost fully consumed. Therefore, P2 only starts pumping when the tank is half full. As for Behaviour B, M only starts mixing when the tank is full to reduce energy consumption.

TABLE II: Descriptions of the sensor and actuator SIFBs

Input and Output	Description
$(P_i, REQ, \{OUT = true\})$	P_i starts pumping.
$(P_i, REQ, \{OUT = false\})$	P_i stops pumping.
$(D, REQ, \{OUT = true\})$	D starts draining.
$(D, REQ, \{OUT = false\})$	D stops mixing.
$(M, REQ, \{OUT = true\})$	M starts draining.
$(M, REQ, \{OUT = false\})$	M stops mixing.
$(S_i, CNF, \{IN = true\})$	The liquid is above S_i.
$(S_i, CNF, \{IN = false\})$	The liquid is below S_i.

The IEC 61499 application describing the initial behaviour of the mixing tank is depicted in Fig. 6. *S1*, *S2*, and *S3* are the sensor SIFBs. *D*, *P1*, *P2*, and *M* are the actuator SIFBs. The inputs and outputs of these SIFBs are described in Table II. The other FBs (e.g., *ED1*, *ES1*, and *TFD*) regulate the interactions between the sensor and actuator SIFBs such that the mixing tank operates according to Behaviour 1.

The following section shows how to generate guidelines to evolve the application in Fig. 6 into applications that execute according to Behaviour A and Behaviour B.

IV. Guided Evolution Techniques

This work aims to guide the evolution of IEC 61499 applications to satisfy the requirements. Our approach relies on the generated behavioural model of the application. We propose algorithms and analysis techniques to infer the modifications in the application using the model and the requirements.

The overview of the approach, which consists of three modules, is presented in Fig. 7. In the *Preliminary* module, the application is translated into a model (Definition II.2), and the requirements are interpreted as a specification. The *Model evolution* module first analyses the generated model and the specification to identify relevant submodels (*Model analysis*). Afterwards, the submodels are modified according to the specification (*Submodels modification*). The last module, *Guidelines generation*, compares the identified and modified submodels to generate guidelines for evolving the application.

A. Preliminary

Our approach takes as input an IEC 61499 application and requirements to be satisfied. The *Preliminary* module preprocesses these inputs before using them to generate the evolution guidelines. The application is translated into a model given in Definition II.2, whereas the requirements are interpreted as a specification explained in this section.

Modelling. The translation techniques to obtain behavioural models of IEC 61499 applications have been proposed in [13]. They are not part of the contribution of this work. Therefore, the details of these techniques are not given in this paper. We use the translation techniques in [13] on the application in Fig. 6 to obtain a model consisting of 49392 states and 340956 transitions. To illustrate, we present a fragment of this model in Fig. 8. It consists of three states and their outgoing transitions. The model shows that the initial state can trigger six possible events. These are events that are triggered by the sensor SIFBs. Furthermore, in state 2, *ED1* can trigger *EO* with $Q = false$ after the value of *IN* in *S1* changes from *true* (transition 0 to 1) to *false* (transition 1 to 2).

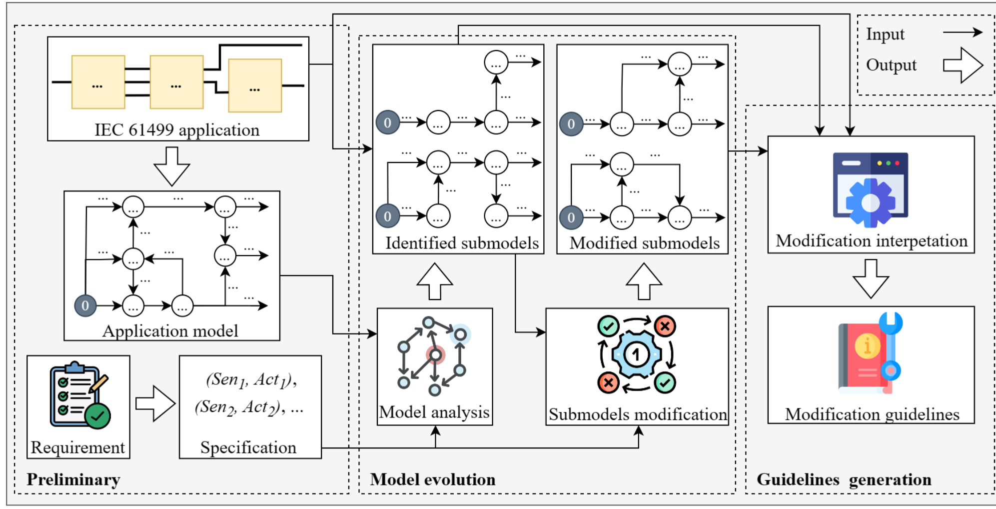


Fig. 7: Overview of the guided evolution techniques

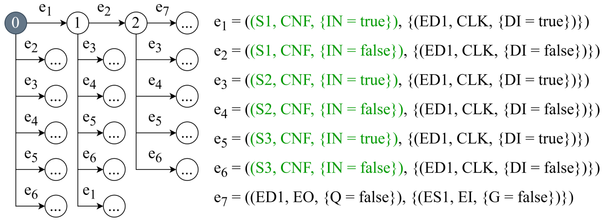


Fig. 8: Fragment of the initial application model

Requirements. In this work, the description of the application behaviour is represented as interactions. An interaction consists of a sequence of outputs triggered by the sensor SIFBs and a set of inputs received by the actuator SIFBs.

Definition IV.1 (Interaction)**.** *An interaction is a tuple (Sen, Act), where $Sen = o_1, o_2, ..., o_n$ is a sequence of outputs triggered by the sensor SIFBs, and $Act = \{i_1, i_2, ..., i_n\}$ is a set of inputs received by the actuator SIFBs.*

The behaviours in Table I can be represented as interactions using the descriptions in Table II. As an example, the first statement of the initial behaviour can be represented as (Sen, Act), where $Sen = (S1, CNF, \{IN = true\}), (S1, CNF, \{IN = false\})$ and $Act = \{(P1, REQ, \{OUT = true\}), (P2, REQ, \{OUT = true\}), (D, REQ, \{OUT = false\})\}$.

The evolution requirements describe the new behaviour of the application. Therefore, these requirements can be represented as a set of new interactions that do not belong to the application's initial behaviour. This set is called a *specification*.

Definition IV.2 (Specification)**.** *A specification is a set of interactions $\{(Sen_1, Act_1), (Sen_2, Act_2), ..., (Sen_n, Act_n)\}$.*

A specification r is obtained by first specifying the evolution requirements. In our example, these are the bold texts in Table I. Afterwards, we use the description of the sensor and actuator SIFBs in Table II to interpret the requirements into a specification.

Table III presents the specifications of Behaviour A and Behaviour B as Specification A and Specification B. Each specification describes the behaviours of the target application that differ from the initial application. For instance, the first point of Behaviour A differs from Behaviour 1 because P2 does not start pumping when the liquid moves below S1. This difference is represented as an interaction (Sen_1, Act_1) in Specification A. Sen_1 contains a sequence of outputs corresponding to the movement of the liquid from above S1 (i.e., $(S1, CNF, \{IN = true\})$) to below S1 (i.e., $(S1, CNF, \{IN = false\})$). Act_1 contains two inputs received by the actuators P1 and D. Here, P1 should start pumping (i.e., $(P1, REQ, \{OUT = true\})$), and D should stop draining (i.e., $(D, REQ, \{OUT = false\})$). Notice that $(P2, REQ, \{OUT = true\}) \notin Act_1$ because P2 should not start pumping when the liquid moves below S1.

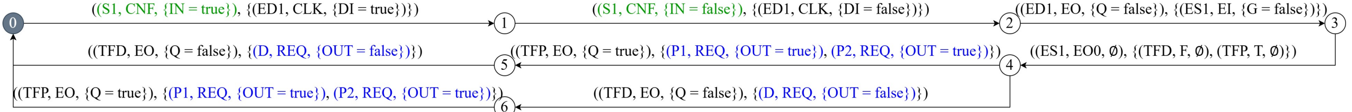
B. Model evolution

The *Model evolution* module generates submodels that can guide the application's evolution. The initial model describes all possible sequences of events that the application can trigger. Some of these sequences are unrelated to the specification. The first step in this module, *Model analysis*, identifies sequences in the model that are relevant to the specification. This allows us to focus on the part of the model that needs to be modified. The second step, *Submodels modification*, modifies the identified submodels according to the specification.

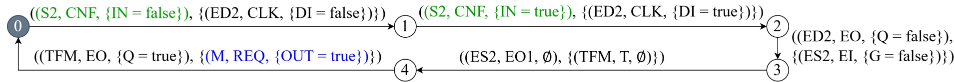
Model analysis. Algorithm 1 describes how relevant submodels are identified. It takes the initial application model m_{init} and the specification r as input to return a set of identified submodels M_{ident}. A submodel consists of event sequences that begin with the events triggered by the sensor SIFBs and end with the events received by the actuator SIFBs. Function *isNext* checks if an event output is the next one in

TABLE III: Specifications of the mixing tank system according to Behaviour A and Behaviour B

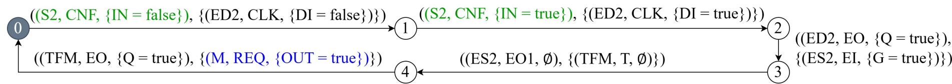
Specification A	Specification B
• (Sen_1, Act_1) $Sen_1 = (S1, CNF, \{IN = true\}),\ (S1, CNF, \{IN = false\})$ $Act_1 = \{(P1, REQ, \{OUT = true\}),\ (D, REQ, \{OUT = false\})\}$ • (Sen_2, Act_2) $Sen_2 = (S2, CNF, \{IN = false\}),\ (S2, CNF, \{IN = true\})$ $Act_2 = \{(P2, REQ, \{OUT = true\}),\ (M, REQ, \{OUT = true\})\}$	• (Sen_1, Act_1) $Sen_1 = (S2, CNF, \{IN = false\}),\ (S2, CNF, \{IN = true\})$ $Act_1 = \varnothing$ • (Sen_2, Act_2) $Sen_2 = (S3, CNF, \{IN = false\}),\ (S3, CNF, \{IN = true\})$ $Act_2 = \{(P1, REQ, \{OUT = false\}),\ (P2, REQ, \{OUT = false\}),$ $(D, REQ, \{OUT = true\}),\ (M, REQ, \{OUT = true\})\}$



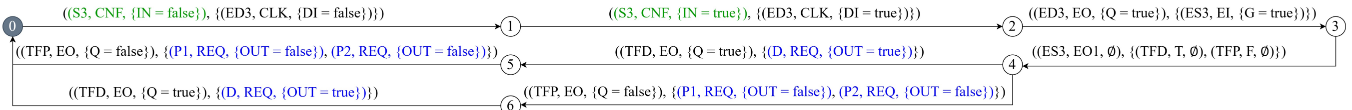
(a) The first identified submodel according to Requirement A (Submodel A1)



(b) The second identified submodel according to Requirement A (Submodel A2)



(c) The first identified submodel according to Requirement B (Submodel B1)



(d) The second identified submodel according to Requirement B (Submodel B2)

Fig. 9: Identified submodels

Algorithm 1: Model analysis

input : $m_{init} = (S, s_0, E, T),\ r = \{(Sen_1, Act_1), (Sen_2, Act_2), ..., (Sen_n, Act_n)\}$
output: $M_{ident} = \{m_1, m_2, ..., m_n\}$

1 **foreach** $(Sen, Act) \in r$ **do**
2 $\quad m' := (\varnothing, s_0, \varnothing, \varnothing)$
3 $\quad$ *identify* $(s_0, m_{init}, m', Sen, \epsilon)$
4 $\quad M_{ident} := M_{ident} \cup m'$

5 **identify** (s_c, m, m', Sen, e_p)
6 $\quad$ **let** T_c be the set of transitions outgoing from s_c **in**
7 $\quad\quad$ **foreach** $t = (s, e, s') \in T_c$, **do**
8 $\quad\quad\quad$ **if** *isNext* $(e, Sen) \vee$ *isTrig* (e, e_p) **then**
9 $\quad\quad\quad\quad$ *addToModel* (m', t)
10 $\quad\quad\quad\quad$ *identify* (s', m, m', Sen, e)

the sequence, whereas function *isTrig* checks if an event is triggered by the FB to which the previous event was sent. Function *addToModel* adds a transition into a submodel. The algorithm iterates through the set of interactions (lines 1 to 4). In each iteration, a submodel is built by traversing the initial model. A transition is added into the model if its event output is the next one in Sen or triggered by the FB to which the previous event was sent (lines 8 and 9).

We apply Algorithm 1 on the initial model of the application to obtain submodels shown in Fig. 9. There are two submodels for each specification because both Specification A and Specification B contain two interactions. Each submodel describes the sequences of events that can be triggered according to the sequence of outputs in an interaction. For instance, in submodel A1 (Fig. 9a), the first two transitions correspond to the sequence of outputs in Sen_1 (see Table III, Specification A). This submodel describes the sequences of events that can be triggered after the value of IN in $S1$ changes from $true$ to $false$. It informs us about the sequences leading to the inputs received by the actuators. One of the inputs is $(P2, REQ, \{OUT = true\})$ (in transitions 4 to 5 and 6 to 0). This input is not in the set of expected inputs (i.e., Act_1) and must be removed by modifying the submodel. This modification is computed in the second step of this module.

Submodels modification. Algorithm 2 describes how the identified submodels are modified according to the specification. It takes the set of identified submodels M_{ident} and the

Algorithm 2: Submodels modification

```
input : M_ident = {m_1, m_2, ..., m_n}, r = {(Sen_1, Act_1),
        (Sen_2, Act_2), ..., (Sen_n, Act_n)}
output: M_modif = {m'_1, m'_2, ..., m'_n}

1  let P be the set of pairs {(m_1, (Sen_1, Act_1)),
   (m_2, (Sen_2, Act_2)), ..., (m_n, (Sen_n, Act_n))} such
   that m_i ∈ M_ident and (Sen_i, Act_i) ∈ r in
2    foreach (m = (S, s_0, E, T), (Sen, Act)) ∈ P, do
3      I_rem := getInputs(m) \ Act
4      I_add := Act \ getInputs(m)
5      remInputs (s_0, m, I_rem)
6      addInputs (s_0, m, I_add)
7      M_modif := M_modif ∪ m

8  remInputs (s_c, m, I_rem)
9    let T_c be the set of transitions outgoing from s_c in
10     foreach t = (s, e, s') ∈ T_c, do
11       I_tmp := e.I ∩ I_rem
12       if checkRem(I_tmp, m) then
           removeTrans (t, m)
13       else modifyTrans (I_tmp, t, m)
14       remInputs (s', m, I_rem)

15 addInputs (s_c, m, I_add)
16   let T_c be the set of transitions outgoing from s_c in
17     foreach t = (s, e, s') ∈ T_c, do
18       if checkAdd(t) then interleave (I_add, t, m)
19       addInputs (s', m, I_rem)
```

specification r as input. The algorithm returns a set of modified submodels M_{modif}. P is a set of pairs. Each pair contains a submodel and its corresponding interaction (e.g., submodel A1 is paired with interaction (Sen_1, Act_1) in Specification A). Function *getInputs* returns a set of inputs received by the actuators of a given submodel. For instance, applying this function to submodel A2 returns $(M, REQ, \{OUT = true\})$.

The algorithm iterates through the set of pairs to modify the identified submodels (lines 2 to 7). In each iteration, it initiates the sets of inputs to be removed and inputs to be added (lines 2 and 3). Function *remInputs* traverses a submodel to find transitions labelled with events containing inputs that need removal (lines 8 to 16). If the event contains another input corresponding to the same connection, the transition is modified by removing only the input in the set I_{tmp} (line 15); otherwise, the transition is removed completely (line 13). Function *addInputs* traverses the submodel for the second time (lines 17 to 22) to integrate the set of inputs I_{add} to the submodel by building interleaving transitions with other transitions containing inputs that the sensor receives (line 21).

Fig. 10 shows the modified submodels obtained using Algorithm 2. Each submodel now contains sequences of events that satisfy the specification. For instance, the input $(P2, REQ, \{OUT = true\})$ is removed in the modified submodel A1 (Fig. 10a) because it is not one of the expected inputs in the corresponding interaction (i.e., Act_1 in Specification A). However, this input is added to the modified submodel A2 (transitions 4 to 6 and 5 to 0 in Fig. 10b) because it is one of the expected inputs in Act_2.

TABLE IV: Modification patterns

	Comparison	Modification
(1)	An input is missing from the set of inputs in the transition.	Remove the connection between the output and the missing input
(2)	An input is added to the set of inputs in the transition.	Create a connection between the output and the new input
(3)	A transition is added or removed.	Modify the ECC of the corresponding FB

C. Guidelines generation

The *Guidelines generation* module analyses the initial and modified submodels to generate evolution guidelines. We propose modification patterns in Table IV to help this analysis. Every identified submodel (e.g., Fig. 9a) and its modified version (e.g., Fig. 10a) are compared to find their differences. Then, the modification patterns are used to find suitable modifications that must be applied to the application according to the comparison.

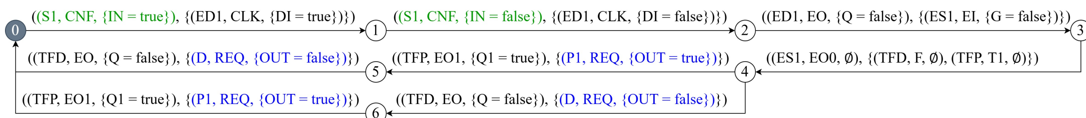
TABLE V: Evolution guidelines

Guidelines A
(i) remove $(ES1, EO0)$ - (TFP, T), (TFP, EO) - $(P1, REQ)$, and (TFP, EO) - $(P2, REQ)$
(ii) create $(ES1, EO0)$ - $(TFP, T1)$, $(ES2, EO1)$ - $(TFP, T2)$, $(ES3, EO1)$ - $(TFP, F1)$, $(TFP, EO1)$ - $(P1, REQ)$, and $(TFP, EO2)$ - $(P2, REQ)$
(iii) modify TFP
Guidelines B
(i) remove $(ES2, EO1)$ - (TFM, T)
(ii) create $(ES3, EO1)$ - (TFM, T)

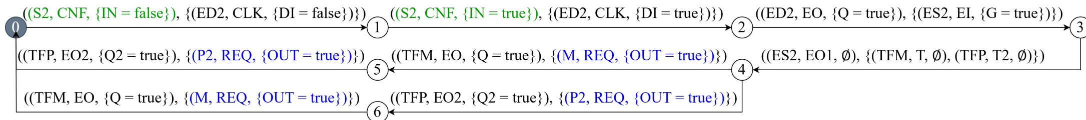
The evolution guidelines to obtain applications with Behaviour A and Behaviour B are presented in Table V as Guidelines A and Guidelines B. Here, $(Source\ FB, Interface)$ - $(Target\ FB, Interface)$ denotes a connection between two FBs. The guidelines are generated according to the comparison between submodels in Figs 9 and 10. For instance, in Guidelines B (i), the connection between $EO1$ in $ES2$ and T in TFM must be removed because the input $(TFM, T, \varnothing)$ is removed in the modified submodel B1. On the other hand, the connection between $EO1$ in $ES3$ and T in TFM (i.e., Guidelines B (ii)) must be created because the input $(TFM, T, \varnothing)$ is added in the modified submodel B2 (transition 3 to 4). The fragments of the evolved applications according to the guidelines are shown in Fig 11.

V. Implementation

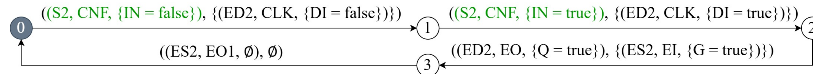
We have developed a prototype to automate the *Model evolution* module. It is written in Java and takes two text files as input. The first file is the textual representation of the



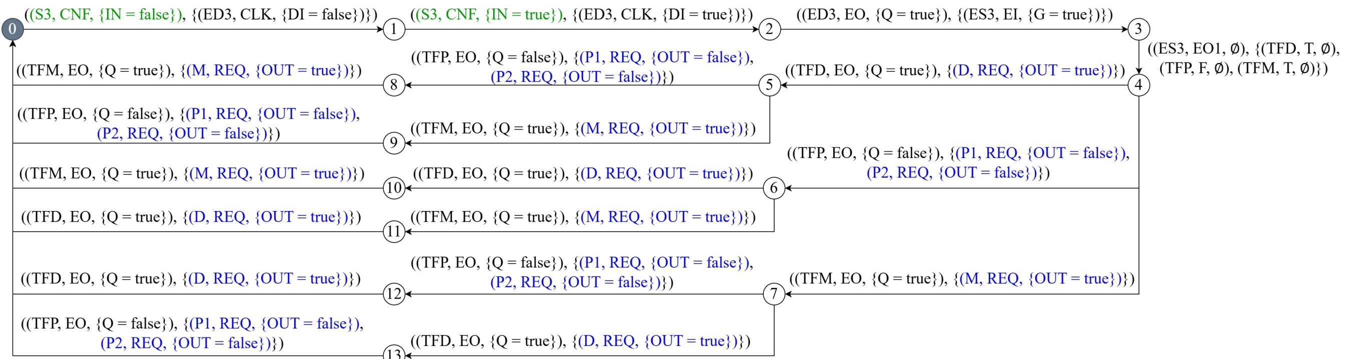
(a) Modified A1



(b) Modified A2

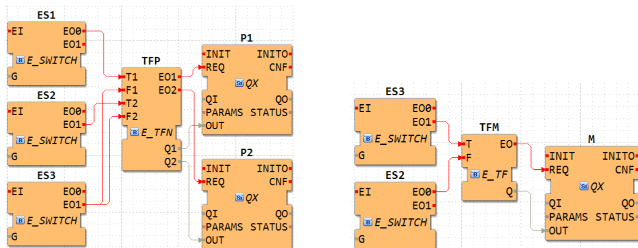


(c) Modified B1



(d) Modified B2

Fig. 10: Modified submodels



(a) According to Guidelines A (b) According to Guidelines B

Fig. 11: Fragments of the evolved applications

```
=== Interaction 1 ===
Outputs sequence: (S2, CNF, !FALSE), (S2, CNF, !TRUE)
Expected inputs: {}

--- Identified LTS ---
Source: 0
  o: (S2, CNF, !FALSE), I: {(ED2, CLK, !FALSE)}, Target: 45576
Source: 688
  o: (ES2, EO1, ), I: {(TFM, T, )}, Target: 48178
Source: 48178
  o: (TFM, EO, !TRUE), I: {(M, REQ, !TRUE)}, Target: 21734
Source: 21734
Source: 45576
  o: (S2, CNF, !TRUE), I: {(ED2, CLK, !TRUE)}, Target: 49390
Source: 49390
  o: (ED2, EO, !TRUE), I: {(ES2, EI, !TRUE)}, Target: 688

--- Computed modifications ---
To remove: (M, REQ, !TRUE)
To add:

--- Modified LTS ---
Source: 0
  o: (S2, CNF, !FALSE), I: {(ED2, CLK, !FALSE)}, Target: 45576
Source: 688
  o: (ES2, EO1, ), I: {}, Target: 48178
Source: 48178
Source: 45576
  o: (S2, CNF, !TRUE), I: {(ED2, CLK, !TRUE)}, Target: 49390
Source: 49390
  o: (ED2, EO, !TRUE), I: {(ES2, EI, !TRUE)}, Target: 688
```

Fig. 12: Excerpt of the prototype's outputs

application model. The second one is the specification with the same format as in Table. III. It generates the identified and modified submodels in textual format as output.

Fig. 12 shows an excerpt of the prototype's outputs. This prototype considers only one data interface associated with an event interface. Thus, only the value of the data is shown (e.g., $!FALSE$ instead of $\{IN = false\}$). The excerpt shows the identified and modified submodels corresponding to the first interaction of Requirement B (see Figs. 9c and 10c).

Every module in our approach can be automated. However, associating the input and output SIFBs with physical activities requires manual intervention (e.g., Table II). Nevertheless, this is not an issue because SIFBs include documentation explaining their interfaces. Furthermore, our approach should scale well for large applications because the modification algorithm is applied to submodels with much smaller sizes (e.g., seven states in Submodel A1) than the initial model (e.g., 49392 states in the running example model).

VI. RELATED WORK

A featured-oriented evolution method for automation systems is proposed in [19]. It relies on the mapping between features and codes. The evolution is simplified by automatically updating the code when a new feature is introduced (e.g., a second drill). Compared with our work, this approach is useful for adding and removing features, while ours helps to modify existing features according to the requirements.

The authors in [6] propose a downtimeless evolution method for IEC 61499 applications. The method relies on a new type of FB called EvoFB, which encapsulates three sequences: initiations (*RINIT*), reconfiguration (*RECONF*), and termination (*RDINIT*). The follow-up works in [7], [8] propose verification techniques to evaluate the correctness of the reconfiguration process. Their method is extended in [9] for distributed applications. The work in [10] generates the correct order of the reconfiguration operations (i.e., *RECONF*) using dependency graphs. This approach is refined in [11] to support real-time systems by considering the timing constraints and applying the priority ceiling protocol. This collection of works (i.e., [6]–[11]) facilitates the seamless evolution of IEC 61499 applications without stopping their executions. In comparison, our approach focuses on finding the target application for a given initial application and evolution requirements. The method proposed in this paper can be integrated into the aforementioned downtimeless evolution techniques to build an end-to-end framework for evolving IEC 61499 applications.

The work in [20] proposes an approach to automatically reconfigure the mappings of FBs in IEC 61499 applications into the control devices according to requirements. Applications are represented as algebraic models describing the mappings between the FBs and the control devices. Requirements are expressed using quantifier-free first-order formulae. A configuration engine based on SMT constraint resolution computes the system's satisfactory configuration. Both this work and ours aim to support the evolution of IEC 61499 applications. However, they focus on the application's structural aspect when specifying the target evolution. In contrast, we deal with the behavioural aspect of the application.

Runtime enforcement for IEC 61499 applications is proposed in [21]. This work integrates an enforcer FB into the application to modify its runtime behaviour. Properties are expressed as state machines. Transitions can be specified to forward, discard, and replace events. For instance, an event to start the mixing process can be replaced with another event that starts the draining process. This work can also be used to evolve IEC 61499 applications according to requirements. However, every evolution involves the addition of an enforcer FB. In comparison, our approach does not add new FBs to the application, which avoids adding complexity.

VII. CONCLUSION

This work proposes a set of techniques to guide the evolution of IEC 61499 applications according to some requirements. These techniques make use of various algorithms and analysis techniques on the behavioural model of the application to generate evolution guidelines. These guidelines help to make the necessary modifications without adding extra complexity or creating erroneous behaviour. For future work, we plan to integrate our approach with existing tools and frameworks that can evolve IEC 61499 applications without stopping their executions.

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] G. Taentzer, M. Goedicke, B. Paech, K. Schneider, A. Schürr, and B. Vogel-Heuser, "The nature of software evolution," in *Managed Software Evolution*. Springer, 2019, pp. 9–20.
[2] V. E. S. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution," *Comput. Sci. Res. Dev.*, vol. 28, no. 4, pp. 311–329, 2013.
[3] M. M. Lehman, "Laws of software evolution revisited," in *Proc. of EWSPT'96*, ser. LNCS, vol. 1149. Springer, 1996, pp. 108–124.
[4] "International Electrotechnical Commission, Functional blocks - Part 1: Architecture, 2nd edn, IEC 61499-1," *IEC Geneva*, 2012.
[5] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Trans. Ind. Informatics*, vol. 7, no. 4, pp. 768–781, 2011.
[6] O. Hummer, C. Sünder, A. Zoitl, T. I. Strasser, M. N. Rooker, and G. Ebenhofer, "Towards zero-downtime evolution of distributed control applications via evolution control based on IEC 61499," in *Proc. of ETFA'06*. IEEE, 2006, pp. 1285–1292.
[7] C. Sünder, B. Favre-Bulle, and V. Vyatkin, "Towards an approach for the verification of downtimeless system evolution," in *Proc. of ETFA'06*. IEEE, 2006, pp. 1133–1136.
[8] C. Sünder, V. Vyatkin, and A. Zoitl, "Formal verification of downtimeless system evolution in embedded automation controllers," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1, pp. 17:1–17:17, 2013.
[9] A. Schimmel and A. Zoitl, "Distributed online change for IEC 61499," in *Proc. of ETFA'11*. IEEE, 2011, pp. 1–7.
[10] L. Prenzel and S. Steinhorst, "Automated dependency resolution for dynamic reconfiguration of IEC 61499," in *Proc. of ETFA'21*. IEEE, 2021, pp. 1–8.
[11] L. Prenzel, S. Hofmann, and S. Steinhorst, "Real-time dynamic reconfiguration for IEC 61499," in *Proc. of ICPS'22*. IEEE, 2022, pp. 1–6.
[12] A. Zoitl and R. Lewis, *Modelling control systems using IEC 61499. 2nd Edition*. Institution of Engineering and Technology, 2014.
[13] Y. Falcone, I. Faqrizal, and G. Salaün, "Probabilistic analysis of industrial IoT applications," in *Proc. of IoT'22*. ACM, 2022, pp. 41–48.
[14] I. Faqrizal, T. Liakh, M. Xavier, G. Salaün, and V. Vyatkin, "Probabilistic model checking for IEC 61499: A manufacturing application," in *Proc. of ICIT'24*. IEEE, 2024, pp. 1–6.
[15] "Programmable controllers-part 3: Programming languages," *IEC 61131-3 (Ed. 2.0)*, 2002.
[16] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, and A. Martel, "Framework for distributed industrial automation and control (4diac)," in *Proc. of INDIN'08*, 2008, pp. 283–288.
[17] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.
[18] T. Chen and C. Cheng, "Modelling and verification of an automatic controller for a water treatment mixing tank," *Desalination and Water Treatment*, vol. 159, pp. 318–326, 2019.
[19] D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, and A. Egyed, "Feature-oriented evolution of automation software systems in industrial software ecosystems," in *Proc. of ETFA'18*. IEEE, 2018, pp. 107–114.
[20] R. Sinha, K. Johnson, and R. Calinescu, "A scalable approach for re-configuring evolving industrial control systems," in *Proc. of ETFA'14*. IEEE, 2014, pp. 1–8.
[21] Y. Falcone, I. Faqrizal, and G. Salaün, "Runtime enforcement for IEC 61499 applications," in *Proc. of SEFM'22*, ser. LNCS, vol. 13550. Springer, 2022, pp. 352–368.