Probabilistic Analysis of Industrial IoT Applications

Yliès Falcone, Irman Faqrizal, Gwen Salaün

Probabilistic Analysis of Industrial IoT Applications

Yliès Falcone
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
38000 Grenoble, France

Irman Faqrizal
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
38000 Grenoble, France

Gwen Salaün
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
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 challenging because of their informal semantics and distributed logic. We introduce new verification techniques for IEC 61499 applications that combine a design-time analysis for computing a formal model of the application with a runtime analysis for extracting additional information during the execution. This combination of analyses allows for building a richer model of the application and thus formally verifying it using probabilistic model checking. Our approach is automated using several tools and was validated on a realistic IEC 61499 application.

KEYWORDS

Industrial IoT, IEC 61499, Probabilistic model checking, Runtime verification

ACM Reference Format:
Yliès Falcone, Irman Faqrizal, and Gwen Salaün. 2022. Probabilistic Analysis of Industrial IoT Applications. In *Proceedings of the 12th International Conference on the Internet of Things (IoT '22), November 7–10, 2022, Delft, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3567445.3567461

1 INTRODUCTION

The industrial Internet of Things (IoT) incorporates interconnected smart devices in the industrial setting to achieve specific objectives. One of its essential elements is the industrial automation. It uses distributed control devices to automate the component processes in a system. Such control devices, e.g. Programmable Logic Controllers (PLCs), can be programmed individually using a conventional standard IEC 61131-3. However, the standard does not specify how PLCs should communicate; thus, issues may arise when PLCs originate from different vendors. Moreover, such an approach is fading because the industry is moving towards a more distributed paradigm. The International Electrotechnical Commission (IEC) 61499 [3] is a recent standard for developing industrial automation. This standard 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.

IoT '22, November 7–10, 2022, Delft, Netherlands

The main benefit of IEC 61499 is that it is suitable for developing a fully distributed system [19]. A system that consists of several control devices can be designed as a whole application. 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. Furthermore, the execution environment heavily influences the behavior of industrial IoT applications. In contrast to conventional programs with mostly user interactions, industrial applications 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. The only scalable approach for analysing such systems is to run and monitor their executions. Ultimately, verifying the correctness of an IEC 61499 application is not straightforward since the standard has loosely defined semantics [18].

There exist several works that propose techniques for verifying IEC 61499 applications. First, the applications can be verified during design time using static verification techniques such as model checking [1, 15]. Such an approach is useful for finding unexpected behaviour before the application deployment. However, as previously mentioned, industrial applications often interact with nondeterministic behaviours coming from the environment, which can not be observed during design time. The approaches in [4] and [12] propose alternative methods to verify IEC 61499 applications by applying runtime verification techniques [6]. The approach involves analyses of execution traces obtained directly from executing the application (considering the influence of the environment). However, runtime analysis only considers observable behaviours when executing applications; it is not as exhaustive as static verification techniques. Altogether, new techniques are required to analyse IEC 61499 applications. These techniques should be able to verify the correctness of an application while also taking into account its runtime behaviour.

In this work, we propose to combine design time and runtime analyses to verify industrial IoT applications described with the IEC 61499 framework. Firstly, at design time, all possible executions of the application (under verification) are captured using a behavioural model, namely Labelled Transition System (LTS). This model is obtained by translating the IEC 61499 application into a specification describing the application behaviour, from which we can generate an LTS. Next, we instrument the application to insert a monitor as a function block, which allows executing a monitored version of the verified system with the existing IEC 61499 runtime environments. Once the application executes, the monitor can observe and store the execution traces of this application. Then, we enrich the generated model (i.e., LTS) with the probabilistic values computed from these executions. As a result, we obtain a probabilistic model. Finally, we can use probabilistic model checking to verify certain requirements described as probabilistic properties.

To summarise, the contributions of this work are as follows:

(i) an automated transformation technique from a given IEC 61499 application into a behavioural model,
(ii) a monitoring method to extract execution traces from a running application,
(iii) an algorithm to compute the probabilistic model from the given model and traces.

Several tools are implemented to automate the approach, and a realistic example is presented to demonstrate the techniques.

The paper is organised as follows. Section 2 introduces background notions. Section 3 discusses the details of our approach. Section 4 presents the supporting tools. Section 5 reports on a case study. Section 6 surveys related work and Section 7 concludes.

2 BACKGROUND

We first recall some models used in our approach (Section 2.1) and then present the necessary concepts from IEC 61499 (Section 2.2).

2.1 Models

A Labelled Transition System (LTS) is a 4-tuple (S, s^0, Σ, T), where S is the set of states, $s^0 \in S$ is the initial state, Σ is the set of actions, and $T \subseteq S \times \Sigma \times S$ is the transition relation between states labelled by actions. An LTS is deterministic whenever T is a function. For a deterministic LTS, an execution is a sequence of transition actions, traversed by starting from the initial state and following the transition function.

A Probabilistic Transition System (PTS) is an LTS where transitions are extended with probabilities. That is, the transition relation is a subset of $S \times \Sigma \times p \times S$, where each $p \in [0, 1]$ is the probability of executing a transition from its source state. The sum of probabilities of transitions outgoing from a given state is equal to 1.

LOTOS New Technology (LNT) is an extension of LOTOS, an ISO standardised process algebra [11], which defines data types, functions, and processes. LNT processes are built from actions, choices (select), parallel composition (par), looping behaviours (loop), conditions (if), and sequential composition (;). The communication between processes is carried out by rendezvous on a list of synchronised actions included in the parallel composition (par). Thanks to the LTS semantics of process algebras, one can use the rich set of tools for LTS-based verification thereafter. We use the CADP toolbox [9], which takes as input an LNT specification and generates the corresponding LTS. CADP is also used for analysing that LTS using classic or probabilistic model checking techniques.

2.2 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. 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.

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 event and/or execute an algorithm written in structured text. The transitions in an ECC can be guarded by some conditions on the data interfaces. A composite FB is composed of a network of FBs. A service interface FB concerns FBs that are specific to their control devices (i.e., vendor dependent). We do not consider service interface FBs since they often have undefined behaviour.

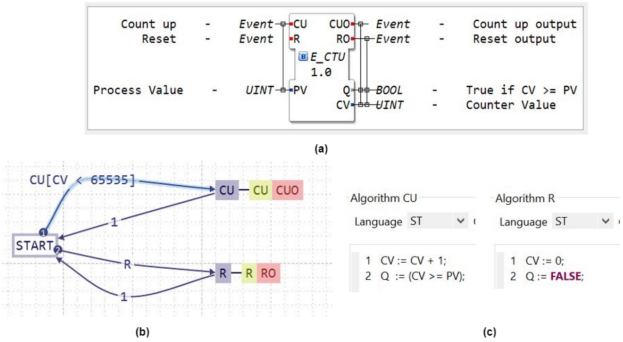


Figure 1: (a) E_CTU FB, (b) ECC, (c) Algorithms.

In our work, we use 4DIAC-IDE [17] as a development environment, an open source tool to build IEC 61499 applications. Figure 1 shows an example of basic FB represented with 4DIAC-IDE. This FB is called *E_CTU* and behaves as a counter. As shown in Figure 1 (a), there are two input event interfaces (*CU* and *R*), an input data interface (*PV*), two output event interfaces (*CUO* and *RO*), and two output data interfaces (*Q* and *CV*). The ECC in Figure 1 (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 1 (c)) is executed and output event *CUO* is fired. The algorithm increments *CV* and updates the value of *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 applies when the FB receives event *R*, which resets the counter value *CV*. The work in [16] contains the details of ECC's semantics, which is formalised using Operation State Machine (OSM).

3 PROBABILISTIC MODEL

This section describes how to build a probabilistic model corresponding to an IEC 61499 application.

3.1 Transformation from IEC 61499 Application to LTS

The goal of the first step is to produce a formal model, in the form of an LTS, given an IEC 61499 application. For this, the application is first translated into the LNT specification language. Afterwards, the CADP toolbox compiles the LNT specification and generates the corresponding LTS. LNT is used as the intermediate language to simplify the method by first allowing the translation of individual

FBs to LNT processes which are then combined together using the LNT parallel and synchronisation mechanism.

Note that the resulting LTS model abstracts away some details of the IEC 61499 application. In particular, we do not model the values of the data interfaces, which are not necessary for obtaining the LTS model serving as a semantic model of the IEC 61499 application. Consequently, we do not consider ECC's guarded transitions since they are associated with data values. Discarding data interfaces also permits avoiding the state-space explosion problem that may appear during the LTS generation. The translation from IEC 61499 to LNT is divided into two parts: (i) Translation of each basic FB into an individual process, (ii) Translation of the application's architecture into the main process.

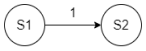
Translation of Function Blocks. Translating a FB into an LNT process relies on the event interfaces and the ECC. The event interfaces are used to define the process parameters, while the ECC is used to write the behaviour of the process. To translate ECC, we represent states as LNT processes and events as actions. Then, we make use of the ECC semantics described in OSM [16] to identify representative translation patterns shown in Table 1. ECCs presented in the table (first column) can be used as building blocks to build any ECC. Each of them is translated to an LNT specification (second column). Translation patterns are as follows.

(1) No event is triggered when an empty transition (i.e., transition labelled with 1) is traversed. Therefore, process *S1* which represents state *S1* is composed of action *i* (denoting an unobservable internal action in LNT) followed by a call to process *S2* which represents the target state.
(2) A sequence of transitions with input events is interpreted as a chain of processes calls where processes represent states. In each process, the input event is represented as an action to be executed just before the process call.
(3) Transitions outgoing from the same state are translated into a non-deterministic choice. In LNT, this is written using the *select* construct where in each choice, the transition and the target state are translated as in pattern (2).
(4) Output events are triggered when a state associated with those events is visited. Therefore, in the translated process, all the output events are represented as actions to be executed just before the process call.
(5) A loop in ECC is translated to LNT as a call to a process representing an already visited state.

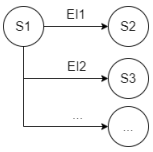
Figure 2 presents an example of the translation of ECC to LNT process. In this example, there are 3 input events (*EI1*, *EI2*, *EI3*) and 3 output events (*EO1*, *EO2*, *EO3*). These events appear as the parameters of the LNT process (lines 1 and 2). The FB name *FB1* is used as the process's name and also as the prefix of subprocesses names (e.g., *FB1_START*) to avoid having processes with the same name. Subprocesses have the same parameters as *FB1* process; for the sake of space, they are hidden. The translation patterns in Table 1 are then used to build the LNT specification. Note that in some specific cases the translation patterns should take into account the connections between FBs. In particular, when there is an input event interface which receives connections from multiple output event interfaces. We illustrate an application with such a situation in Section 5.

Table 1: Translation patterns.

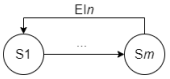
ECC	LNT

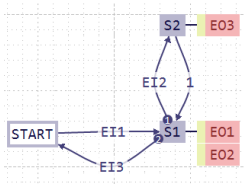
 (1) Empty transition	`process S1 [ ] is`
	`  i ; S2 [ ]`
	`end process`

 (2) Sequential transitions with input events	`process S1 [EI1, EI2, ...] is`
	`  EI1 ; S2 [EI1, EI2, ...]`
	`end process`
	`process S2 [EI1, EI2, ...] is`
	`  EI2 ; S3 [EI1, EI2, ...]`
	`end process`
	`process S3 ...`

 (3) Branching transitions	`process S1 [EI1, EI2, ...] is`
	`  select`
	`  EI1 ; S2 [EI1, EI2, ...]`
	`  []`
	`  EI2 ; S3 [EI1, EI2, ...]`
	`  [] ...`
	`  end select`
	`end process`

 (4) State with output events	`process S1 [EI1, EI2, ...] is`
	`  EI ; EO1 ; EO2 ; ... ; EOn;`
	`  S2 [EI1, EI2, ...]`
	`end process`

 (5) Loop	`process S1 [..., EIn] is`
	`  ... ; ... [..., EIn]`
	`end process`
	`process Sm [..., EIn] is`
	`  EIn ; S1 [..., EIn]`
	`end process`



(a) ECC

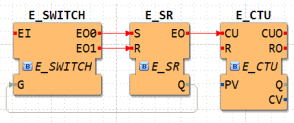
```
1  process FB1 [EI1, EI2, EI3,
2    EO1, EO2, EO3: any] is
3   FB1_START [..]
4  end process
5
6  process FB1_START [..] is
7   EI1 ; EO1 ; EO2 ; FB1_S1 [..]
8  end process
9
10 process FB1_S1 [..] is
11  select
12   EI2 ; EO3 ; FB1_S2 [..]
13  [] EI3 ; EO1 ; EO2 ; FB1_START [..]
14  end select
15 end process
16
17 process FB1_S2 [...] is
18  i ; EO1 ; EO2 ; FB1_S1 [...]
19 end process
```

(b) LNT process and subprocesses

Figure 2: Example of ECC and the corresponding LNT process.

Translation of the Application Architecture. We translate the architecture of the IEC 61499 application into an LNT main process. The translation consists of the following steps:

(1) Analyse the existing FBs to get their names, the interfaces, and the connections.
(2) Create an action for each interface without connection and an action for each output interface connected with input interface(s); then, use these actions as parameters for the main process.
(3) Create a *par* construct which is composed of the calls to the translated processes; then, add parameters and synchronisations according to the FBs' interfaces and connections, respectively.



(a) IEC 61499 Application

```
1  process main [EI, EO0_S, EO1_R,
2    EO_CU, R, CUO, RO: any] is
3   par
4    EO0_S, EO1_R ->
5    E_SWITCH[EI, EO0_S, EO1_R]
6   || EO0_S, EO1_R, EO_CU ->
7    E_SR[EO0_S, EO1_R, EO_CU]
8   || EO_CU ->
9    E_CTU[EO_CU, R, CUO, RO]
10  end par
11 end process
```

(b) LNT main process

Figure 3: Example of IEC 61499 application and the corresponding LNT main process.

Figure 3 shows an example of the translation from IEC 61499 application architecture into an LNT main process. The process parameters consist of all the FBs' event interfaces (lines 1 and 2). FBs are represented as elements of the parallel composition, where each element consists of a process call and synchronisations. As an example, *E_SWITCH* (line 5) and *E_SR* (line 8) processes represent FBs with the same name, and both are synchronised on *EO0_S* and *EO1_R* to represent the connections.

LTS Generation. Given the generated LNT specification, the CADP tool is utilised to compile that specification and obtain the corresponding LTS. The generated LTS may have transitions in some states that yield non-deterministic behaviour. In such a case, the LTS must be transformed so that the process exhibits deterministic behaviour from all states, which will simplify the construction of the PTS presented in Section 3.3. To make the LTS deterministic, we use classic minimisation and determinisation algorithms [10].

Although we mainly focus on probabilistic model checking in this paper, one can also leverage the transformation to LNT and LTS to verify temporal properties of interest (e.g., the absence of deadlocks) using classical model checking techniques available in CADP.

The IEC 61499 standard has loosely defined semantics [18]. This transformation from IEC 61499 to LTS provides the application with a formal model. To ensure that the model represents the actual behaviour of the application, we make use of the following testing method: (1) the IEC 61499 application is first transformed to an LTS, (2) the application is simulated using 4DIAC-IDE for observing its execution traces, (3) we check that the generated traces are also traces of the LTS. Note that the second step makes use of monitoring techniques discussed in the next section.

3.2 Monitoring

The main purpose of the monitoring techniques is to observe the application during its execution and store the so-called execution trace, which will later be used to compute the probabilistic model. To do so, the application needs to be instrumented by inserting a new FB, which is connected to the outputs of all FBs present in the IEC 61499 application and thus acts as a monitor for this application. This section describes how exactly a monitoring FB is added to the application. Note that the monitoring FB will be connected only to the event interfaces since we do not model the values of the data interfaces. The generated traces of our monitoring techniques are sequences of events being executed by the FBs.

The monitoring techniques of an IEC 61499 application consist of the following steps:

(1) Analyse the existing FBs and their connections to obtain the number of interfaces to be monitored along with their names.
(2) Create a monitoring FB with: (i) input event interfaces, (ii) an output interface, (ii) output data interfaces *TRACE* and *INDEX*.
(3) Create connections between the monitoring FB input event interfaces with the existing FBs.
(4) Modify the monitoring FB's code such that when an event is received, an action is recorded in the *TRACE*.

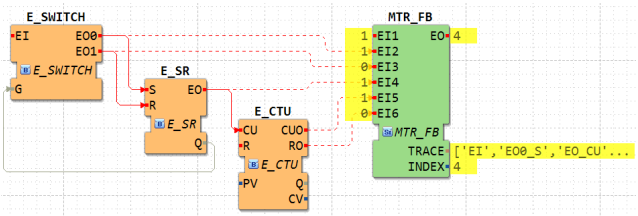


Figure 4: Example of monitored application.

An example of IEC 61499 application extended with a monitor is presented in Figure 4. A monitoring FB called *MTR_FB* is integrated into the system. This FB is synthesised by analysing the FBs in the application. There are six output events to be monitored, hence, the monitoring FB has six input event interfaces (namely, *EI1*, *EI2*, . . ., *EI6*). Each input interface of the monitoring FB is dedicated to the monitoring of the corresponding output event interface of the FBs in the application (e.g., *EI2* is dedicated to monitoring *EO0*). *EI* is connected to a particular FB called *START*, which is specified for triggering the start of the application. Hence, in order to monitor *EI*, *EI1* is connected to the output interface of *START* which is not visible in the application design. Furthermore, the monitoring FB is configured into a different controller device in order to make sure that there will be no impact on the existing device. Note that the monitoring FB is not part of the intended application; its only role is to record the trace by monitoring the events when the application is executing. Therefore, the monitoring FB is excluded from the transformation of IEC 61499 application into LTS (Section 3.1).

Each action in the trace can contain the execution of both input and output events. For example, when *EO0* executes an event, the

action *EO0_S* is recorded into the execution trace. This is because when the monitored output event interface executes an event (e.g., *EO0*), it is assumed that the connected input event interface simultaneously executes an event as well (e.g., *S*). The trace is stored in the output data interface called *TRACE* which is an array of strings. There is also an output data interface *INDEX* to keep track of the number of actions in the trace. By default, the array size is limited to 2000 elements. However, this is not a limitation in the approach since it is possible to add more interfaces to store the trace (e.g., TRACE2, TRACE3, etc). For larger applications, several interfaces may be required to record a trace with a large number of actions.

It is also possible to leverage the execution trace collected by the monitoring techniques to perform runtime verification [7]. To do so, we also need a property as input. This property is usually described as an automaton consisting of states and transitions labelled with events; each state is classified to be either correct or incorrect as described in [8]. The monitoring techniques are then extended with a decision procedure, which can compare the execution trace with this property automaton and thus compute a *verdict*. This verdict is true as long as the current state of the property automaton is a correct state. Otherwise, it becomes false.

3.3 Computing a Probabilistic Model

The probabilistic model is the LTS extended with probabilistic values on its transitions. Such a model is also called Probabilistic Transition System (PTS). A PTS is generated by considering the original LTS and the execution traces obtained using the monitoring techniques. An input trace should have a sufficient length to compute a meaningful probabilistic model. Users may also want to specify that every possible event occurred at least once in the trace. However, this requirement can be applied only under specific circumstances since some events may never be triggered.

Algorithm 1: Computation of PTS

Inputs : $M = (S,\ s^0,\ \Sigma,\ T),\ A \in \Sigma^*$
Output : $M_p = (S_p,\ s_p^0,\ \Sigma,\ T_p)$

1 $s_p^0 := (s^0, 1)$; /* set initial state's counter to 1 */
2 $S_p := \{(s, 0) \mid s \in S \setminus \{s^0\}\} \cup \{s_p\}$; /* set remaining state counters to 0 */
3 $T_p := \{(s, a, s', 0, 0.) \mid (s, a, s') \in T\}$; /* set transition counters and probabilities to 0 */
4 $s_p := s_p^0$; /* current state */
5 **foreach** $a \in A$, *in order* **do**
6 **let** $(s, x) = s_p$ **in**
7 **let** $(s, a, s', y, p) \in T_p$ be the transition from state s on label a **in**
8 $y := y + 1$;
9 $p := y \div x$;
10 **let** $(s', x') \in S_p$ be the successor of (s, x) according to T_p **in**
11 $s_p := (s', x' + 1)$;
12 **end**
13 **return** M_p;

Algorithm for computing PTS. Algorithm 1 takes as inputs an LTS M and a trace $A \in \Sigma^*$ (i.e., a list of actions), while it produces as output a PTS M_p over the same set of labels. The first four lines show the preliminary initialisation of variables. In a PTS, each state $s \in S$ is extended with a counter, whereas each transition $(s, a, s') \in T$ has a counter and a probability. Counters represent how many times states/transitions have been traversed. They are initialised to 0 except for the initial state s^0, which is initialised to 1 because it is the state where the algorithm starts its traversal. The main idea of the algorithm is to compute the probability p from the frequency of visiting the transitions and states. The counters are used for computational purposes and can be removed from the PTS after the computation completes.

More precisely, the PTS is traversed by iterating over the trace starting from the first action (line 5). In each iteration, the corresponding transition from the current state s_p is selected by matching its label with the current action a (lines 6 and 7), Afterwards, the probability of that transition is updated (lines 8 and 9). Then, the target state of the selected transition is assigned to the current state, and its counter is incremented (lines 10 and 11). These steps are repeated until there is no more action in the list. The time and memory complexities of this algorithm are $O(|A|)$ and $O(|S| + |T|)$, respectively.

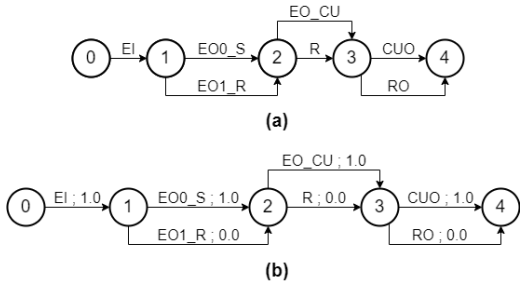


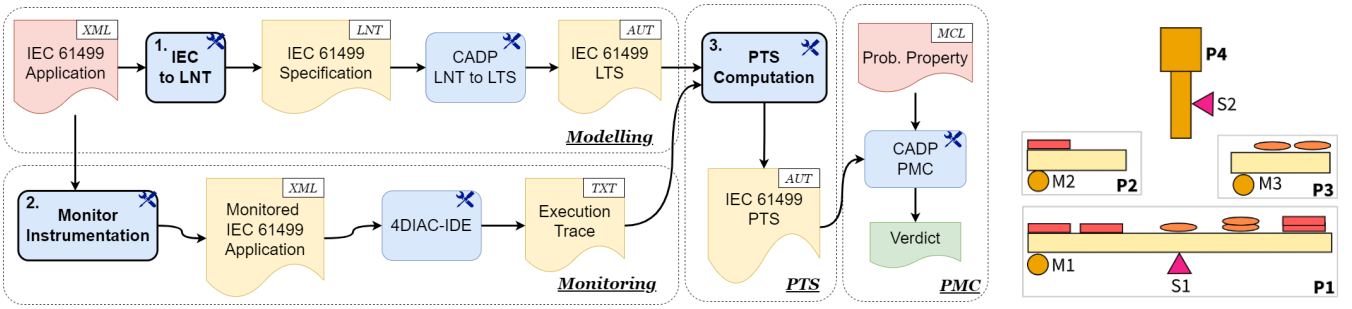
Figure 5: Example of an LTS (a) and computed PTS (b).

As an example, Figure 5 (a) is an excerpt of LTS generated from IEC 61499 application in Figure 4 (without *MTR_FB*). After the application is executed, the monitoring FB returns the trace: *EI*, *EO0_S*, *EO_CU*, *CUO*. Algorithm 1 takes this trace as an input to compute the PTS given in Figure 5 (b). Transitions *EO0_S* and *EO1_R* have probabilities 1.0 and 0.0 because the associated source state (state 1) is visited once, and event *EO0_S* is executed once, while *EO1_R* is never executed.

4 TOOL SUPPORT

Figure 6 (a) overviews the toolchain supporting our approach. We reuse CADP [9] for transforming LNT to LTS, minimising LTSs and probabilistic model checking. Numbered boxes indicate the new tools that we implemented, that is, the transformation from IEC to LNT (1), monitoring techniques (2), and PTS Computation (3). New tools were developed using the Java programming language.

More precisely, the translation from IEC 61499 applications to LNT first needs to read a description of an application encoded as an XML file and returns as output an LNT file. Next, the monitoring techniques automatically extend the application with a monitoring



(a) Overview of the tool support.

(b) Capping station case study.

Figure 6: Tool support and case study.

function block by parsing and modifying its XML file. The type of the monitoring FB is service interface FB and the code is written in C++ programming language. Note that C++ is used since it is the default programming language in 4DIAC-IDE to define the behaviour of a service interface FB. The PTS computation tool takes as input the LTS computed by CADP and the execution traces obtained by simulating the monitored IEC 61499 application using 4DIAC-IDE. Finally, the CADP probabilistic model checker takes as input the computed PTS and a given probabilistic property written using the Model Checking Language [14] (MCL), and returns as output a verdict indicating whether the property is violated or not.

While scalability is not the main focus of this work, we have assessed the performance of the toolchain with IEC 61499 applications of various sizes. For instance, when there are more than 1000 FBs in the application, the translation into LNT and then LTS takes less than 1 minute. When the size of the LTS reaches 10^6 states, it takes only a few minutes to compute the PTS. Nevertheless, this time remains reasonable, especially because these tools execute off-line.

5 CASE STUDY

This section illustrates our approach on an example of IEC 61499 application. The case study is a simplified version of the capping station introduced in [20].

Figure 6 (b) shows the design of the application. The capping station's goal is to put the upper parts of the industrial materials from P2 or P3 onto the lower parts at P1 according to their shapes (i.e. rectangle from P2 or ellipse from P3). Motor M1 moves the lower parts from left to right and stops when a material is detected by sensor S1. Next, depending on which type of material, M2 (if rectangle) or M3 (if ellipse) moves the conveyor slightly to put the upper part in position. Afterwards, a Pick and Place (PnP) unit called P4 grabs the upper part and then puts it onto the lower part. There is a pressure sensor S2 at P4 to detect whether this task is done, in which case it triggers motor M1 to start moving the conveyor again.

The IEC 61499 application which implements the above design is shown in Figure 7. It consists of five FBs that correspond to the sensors and actuators of the original design (namely, S1, S2, M1, M2, M3). The names of the FBs follow the units that they represent (e.g., FB *S1* for sensor S1 in unit P1). Meanwhile, the connections between FBs are established based on the interactions of components. For instance, S2 at P4 is the sensor associated with the activation of motor M1 at P1, therefore, *EO* on *S2* is connected to *R* on *M1* because the value of *Q* on *M1* will change when *R* receives an event. In each FB, there is a boolean data output/input interface which indicates the status of the component. As an example, in *S2* there is a boolean data input interface *PERMIT* which represents whether S2 has detected that the upper part has been placed (*true*) or not (*false*).

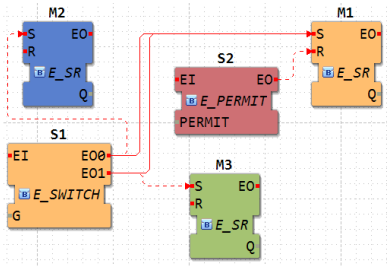


Figure 7: Capping station IEC 61499 application.

Listing 1: LNT main process for the capping station case study.

```
process main[EI, EO0_S, EO1_S, EI_2, EO_R, EO, EO_2,
  EO_3, R, R_2:any] is
 par
  EO0_S, EO1_S -> S1[EI, EO0_S, EO1_S]
 || EO0_S, EO1_S, EO_R -> M1[EO0_S, EO1_S, EO_R, EO]
 || EO0_S -> M2[EO0_S, R, EO_2]
 || EO1_S -> M3[EO1_S, R_2, EO_3]
 || EO_R -> S2[EI_2, EO_R]
 end par
end process
```

Firstly, the application is translated into an LNT specification. In Listing 1, we show only the main process for the sake of space. The interface names in the application are not unique; therefore, some of them are renamed in the LNT specification. For instance, *EO_2* is one of the parameters of process *M2* since the name *EO* is already used by *M1*. Note that in FB *M1*, interface *S* is connected with both *EO0* and *EO1*. In this specific case, every *S* action must be replaced with a *select* construct composed of a choice between *EO0_S* and

EO1_S. This allows us to represent the possibility of triggering *S* by an event coming from either *EO0* or *EO1*.

To observe and store an execution trace, a monitoring FB is inserted into the application. This is done by connecting the output event interfaces of the FBs in the application with the input event interfaces of the monitoring FB as described in Section 3.2. Afterwards, the application is simulated using 4DIAC-IDE. This means that the environment where the application is supposed to interact with is also simulated. In practice, there are two alternatives to simulate the environment. The first one is by triggering the events manually during the simulation, while the second option is to use an additional FB called *E_CYCLE*, which sends events periodically.

The previously generated LTS and the execution trace serve to compute the PTS, which can be used to verify some probabilistic properties using the CADP probabilistic model checker. Examples of properties written in MCL [14] are given in Table 2. For the sake of clarity, we present only properties abiding to the following template:

$$\textbf{prob } R \textbf{ is } \geq\ p \textbf{ end prob}$$

where R is a regular expression representing a set of sequences of actions. The property holds whenever the probability of executing that sequence is greater than or equal to p.

Table 2: Properties verified on the case study.

Property	MCL Definition
φ_1	**prob** $(true^*\ .\ EI\ .\ (\textbf{not}\ EI)^*\ .\ EO\_R)$ **is** $\geq$ 0.9 **end prob**
φ_2	**prob** $(true^*.EI.\ ((EO1\_S.\ EO0\_S)\ \mid\ (EO0\_S.\ EO1\_S)))$ **is** $\leq$ 0.1 **end prob**
φ_3	**prob** $(true^*\ .\ EI\ .\ EO0\_S)$ **is** $\geq$ 0.75 **end prob**

Functional safety properties. An example of abnormal behaviour is that the lower parts exit P1 without the upper parts attached on top of them. Such occurrences are costly since incomplete components can cause failures in the final product of the industrial process; therefore, they should be avoided. Property φ_1 allows detecting this erroneous situation. φ_1 states that every time sensor S1 is triggered (by receiving an event on *EI* at *S1*), S2 will be triggered first (by *EO_R* at *M1*) before S1 is triggered for the second time. This sequence ensures that P4 has finished its job in between the arrival of components at P1. Furthermore, the probability in this property is set to 0.9 (i.e., 90%), meaning that it permits another sequence(s) of actions to happen at most 10% of the time.

Another abnormal behaviour is the activation of both motors M2 and M3 sequentially after sensor S1 is triggered. When this occurs, a lower part component will get both types of attachments. To prevent this, FBs *M2* and *M3* should not be sequentially activated after *S1*. Property φ_2 allows checking such a case by specifying sequences of events that can activate those FBs. It states that event *EI* followed by either *EO1_S* then *EO0_S* or *EO0_S* then *EO1_S* can only happen at most 10% of the time.

Non-functional properties. There are two types of materials in our application (rectangle and ellipse). One may want to check which type of material appears more often than the other. Probabilistic property φ_3 allows detecting such a case. Property φ_3 states that FB *M2* is activated (by receiving event on *EO0_S* after *EI*) more than 75% of the time. This FB represents motor M2 which is triggered whenever sensor S1 has detected a rectangle material. Therefore, we can deduce that whenever this property evaluates to *true*, the rectangle material appears at least 75% of the time.

Evaluating properties. The evaluation of the aforementioned properties depends on the environment, which is equivalent to how the simulation is conducted. This is because we perform simulations to generate execution traces as input for computing the PTS.

Table 3: Verification results of several PTSs.

	M_{p1}	M_{p2}	M_{p3}	M_{p3}	M_{p4}	M_{p5}	M_{p6}	M_{p7}	M_{p8}	M_{p9}	M_{p10}
φ_1	1	1	1	1	1	1	1	1	1	1	0
φ_2	1	1	1	1	1	1	1	1	1	1	1
φ_3	1	0	1	0	0	0	1	0	0	0	0

We have carried out simulations to generate 10 unique execution traces. Table 3 presents the verification results of 10 PTSs computed from those traces. A property satisfaction is denoted by 1 (e.g., M_{p1} satisfies φ_1), whereas 0 signifies a property violation. Property φ_1 is only violated by M_{p10} since the only way to violate the property is by triggering *EI* at least twice before *EO_R*. We purposely performed this several times during the simulation when generating a trace for computing M_{p10}. Meanwhile, φ_2 was always satisfied regardless of how the simulation was conducted because in an *E_SWITCH* FB, *EO0* or *EO1* can only be triggered right after *EI*. Only three PTSs (M_{p1}, M_{p3}, M_{p6}) satisfy φ_3 because the simulations were executed randomly. This implies that the value of *G* in *S1*, which represents the detected component type, has an equal probability to be either *true* (ellipse) or *false* (rectangle). Altogether, these results are useful for observing the impact of the environment on the truth values of the properties.

6 RELATED WORK

The work in [15] 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. The formal model proposed in this work is mostly used for checking functional (qualitative) or behavioural correctness of IEC 61499 applications. Our approach goes further by integrating runtime analysis of the running application, which allows extracting additional information (probabilities), and thus providing a complementary analysis (probabilistic model checking).

Runtime enforcement techniques are used to ensure the correctness of IEC 61499 applications in [5]. Firstly, a property describing the application requirements is specified using an automaton. Next, an enforcer in the form of a basic FB is synthesised from the given property. Lastly, the enforcer is integrated into the application such

that the runtime behaviour follows the specification described in the property. This approach integrate a new FB into the application to change its execution; whereas, our techniques involve the integration of a monitoring FB to record the execution traces.

In [13], the authors define a transformation from an IEC 61499 application to a Business Process Model Notation model. This model presents the automation from a business point of view, and it also enables quantitative analysis of process models. Specifically, it allows business analysts to perform analyses related to cost, resource allocation, and time in automation. This analysis is achieved by transforming the business processes into a formal model in Maude rewriting logic. This is an off-line analysis for IEC 61499, while our focus is to provide verification techniques combining information from the application (design time) and from its execution (runtime).

In [2], a technique to compute a probabilistic model of a stochastic process is proposed. The authors use methods such as frequency estimation and Laplace smoothing to learn probabilistic models based on Discrete Time Markov Chain (DTMC). The main focus is to ensure the global bounds on the error of the models learnt with such techniques by verifying the global behaviour using Computational Tree Logic. An improved learning algorithm is then proposed to respond to the verification result. This approach is achieved using machine learning to build the model, whereas we leverage monitoring techniques to obtain the actual execution trace for computing PTS. Furthermore, the models that we use resemble DTMC. The main difference is that LTS and PTS are event-based models. Whereas, DTMC is a state-based model.

7 CONCLUSION

We have introduced new techniques for analysing industrial applications developed using the IEC 61499 standard. The novelty of this work is that we combine design time and runtime analysis for computing a model of the application. The design time analysis allows the generation of an LTS model, gathering all the possible execution paths of the application. Runtime analysis extracts execution traces for computing probabilities of execution of events, and thus it allows the extension of the LTS model with this quantitative information resulting in a PTS model. These models are then used to carry out model checking on the LTS model, runtime verification on the execution traces, and probabilistic model checking on the PTS model. These checks are complementary and result in the verification of a wide range of properties on IEC 61499 applications. The approach is fully automated and was validated on a realistic case study. According to this case study, we can conclude that our approach is particularly useful in observing how the environment of industrial applications can influence the truth values of probabilistic properties.

One of the possible future works is to take advantage of the model retrieved from the application and the execution trace in order to be able to predict the future behaviour of the application. This would permit anticipating the violations of some properties and thus making decisions to avoid them (e.g., reconfiguration). Another perspective is to consider service interface FBs. In particular FBs that are essential for building IEC 61499 applications, such as for providing interfaces to physical sensors and actuators. This would allow more extensive validation through utilization of the approach for real world industrial examples.

Acknowledgements. This work was partially funded by the D-IIoT project of the LabEx PERSYVAL-Lab (C7H-LXP20A85-D-IIoT).

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.
[2] Hugo Bazille, Blaise Genest, Cyrille Jégourel, and Jun Sun. 2020. Global PAC Bounds for Learning Discrete Time Markov Chains. In *Computer Aided Verif. - 32nd Int. Conf., CAV*, Vol. 12225. 304–326. https://doi.org/10.1007/978-3-030-53291-8_17
[3] International Electrotechnical Commission. 2012. Functional blocks - Part 1: Architecture, 2nd edn, IEC 61499-1. *IEC Geneva* (2012).
[4] Duc Do Tran, Jörg Walter, Kim Grüttner, and Frank Oppenheimer. 2020. Towards Time-Sensitive Behavioral Contract Monitors for IEC 61499 Function Blocks. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, Vol. 1. 27–34. https://doi.org/10.1109/ICPS48405.2020.9274713
[5] Yliès Falcone, Irman Faqrizal, and Gwen Salaün. 2022. Runtime Enforcement for IEC 61499 Applications. In *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022*, Vol. 13550. 352–368. https://doi.org/10.1007/978-3-031-17108-6_22
[6] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. *Engineering Dependable Software Systems* 34 (01 2013), 141–175.
[7] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (2021), 255–284. https://doi.org/10.1007/s10009-021-00609-z
[8] Yliès Falcone and Gwen Salaün. 2021. Runtime Enforcement with Reordering, Healing, and Suppression. In *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021*, Vol. 13085. 47–65. https://doi.org/10.1007/978-3-030-92124-8_3
[9] 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.
[10] John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.
[11] ISO. 1989. *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Technical Report 8807. ISO.
[12] Pranay Jhunjhunwala, Jan Olaf Blech, Alois Zoitl, Udayanto Dwi Atmojo, and Valeriy Vyatkin. 2021. A Design Pattern for Monitoring Adapter Connections in IEC 61499. In *22nd IEEE International Conference on Industrial Technology, ICIT 2021*. IEEE, 967–972. https://doi.org/10.1109/ICIT46573.2021.9453685
[13] Ajay Krishna and Gwen Salaün. 2021. Business Process Models for Analysis of Industrial IoT Applications. In *IoT '21: 11th Int. Conf. on the IoT*. ACM, 102–109. https://doi.org/10.1145/3494322.3494336
[14] Radu Mateescu and Damien Thivolle. 2008. A Model Checking Language for Concurrent Value-Passing Systems. In *15th Int. Symp. on F. Methods*, Vol. 5014. Springer, 148–164. https://doi.org/10.1007/978-3-540-68237-0_12
[15] Polina Ovsiannikova and Valeriy Vyatkin. 2021. 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)*. 01–04. https://doi.org/10.1109/ETFA45728.2021.9613491
[16] Cheng Pang, Sandeep Patil, Chen-Wei Yang, Valeriy Vyatkin, and Anatoly Shalyto. 2014. A portability study of IEC 61499: Semantics and tools. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. 440–445. https://doi.org/10.1109/INDIN.2014.6945553
[17] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Alois Zoitl, Christoph Sünder, Antonio Valentini, and Allan Martel. 2008. Framework for Distributed Industrial Automation and Control (4DIAC). *IEEE Int. Conference on Industrial Informatics (INDIN)*, 283 – 288. https://doi.org/10.1109/INDIN.2008.4618110
[18] 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
[19] 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
[20] Alois Zoitl, Thomas I. Strasser, and Gerhard Ebenhofer. 2013. Developing modular reusable IEC 61499 control applications with 4DIAC. In *11th IEEE Int. Conf. on Ind. Informatics, INDIN*. IEEE, 358–363. https://doi.org/10.1109/INDIN.2013.6622910