THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Techniques de vérification quantitative et d'exécution pour les systèmes d'automatisation industrielle

Quantitative verification and runtime techniques for industrial automation systems

Présentée par :

Irman FAQRIZAL

Direction de thèse :

Gwen SALAUN PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Directeur de thèse
Yliès FALCONE MAITRE DE CONFERENCES, Université Grenoble Alpes	Co-encadrant de thèse

Rapporteurs :

CHRISTIAN ATTIOGBE
PROFESSEUR DES UNIVERSITES, UNIVERSITE DE NANTES
ANTOINE ROLLET
MAITRE DE CONFERENCES HDR, UNIVERSITE DE BORDEAUX

Thèse soutenue publiquement le **5 décembre 2024**, devant le jury composé de :

FABIENNE BOYER, PROFESSEURE DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Présidente
GWEN SALAUN, PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
CHRISTIAN ATTIOGBE, PROFESSEUR DES UNIVERSITES, UNIVERSITE DE NANTES	Rapporteur
ANTOINE ROLLET, MAITRE DE CONFERENCES HDR, UNIVERSITE DE BORDEAUX	Rapporteur
FAROUK TOUMANI, PROFESSEUR DES UNIVERSITES, UNIVERSITE CLERMONT AUVERGNE	Examinateur

Invités :

YLIES FALCONE
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES

Abstract

The rapid advancement in information technology opens up promising opportunities for industrial automation systems. By minimising human intervention, the industry can significantly reduce costs and enhance the system's overall quality. Nevertheless, industrial automation systems are facing many challenges. One of them is how to verify and analyse the quantitative aspects of the system in the presence of an unpredictable environment. Formal methods have been employed recently to verify the system's correctness. However, conventional methods are often applied only at design time, neglecting the runtime execution impacted by the environment. The second major challenge in industrial automation is the need for techniques to support long-running and evolving systems. Manual modification involving human intervention defeats the purpose of automation while also being costly, time-consuming, and erroneous. An automation system must be able to evolve automatically according to requirements.

The main contributions of this thesis are twofold. First, probabilistic model checking is applied to verify and analyse the quantitative aspects of the system originating from the environment. This method consists of formal modelling, monitoring, and probabilistic model computation. The results can be used to observe the impact of the environment and to suggest improvements associated with the system's quantitative features, such as productivity. The second contribution consists of two approaches for evolving automation systems. In the first approach, runtime enforcement techniques are applied to make the application adapt to the requirements. This is done by automatically synthesising and integrating new logical components called enforcers to modify the system's executions according to the requirements. The second proposal incorporates various algorithms applied to the behavioural models of the applications for generating evolution guidelines. These guidelines contain modifications to be applied to make the application satisfy the given requirements. Both solutions allow developers to avoid errors and unnecessary modifications when evolving industrial automation systems. The contributions are aimed at automation systems designed with IEC 61499, a promising industrial standard with numerous positive characteristics. Existing and new software tools are used and developed to conduct case studies and experiments validating the proposed methods.

Contents

1

Introduction

Automation incorporates various technologies to assist human activities by reducing manual intervention. Its history dates back to ancient times when systems such as water clocks and hydraulic automatic doors were invented [Gua10]. In the 18th century, the first industrial revolution marked the beginning of mass production of goods using automation systems based on steam engines [Ros10]. However, it was only in recent years that the transformative power of automation in the industry became more evident than ever before, inspiring a new era of possibilities.

In this era, called Industry 4.0 [Sch17], industrial automation systems are enhanced with advancements in information technology, such as Artificial Intelligence and the Internet of Things [Sis+18]. This results in smart grid [ZV12], smart manufacturing [Li+18], smart factory [RM23], and other advanced applications. These advancements allow automation systems to be more productive, thus contributing to the global economy and asserting their importance [Kro+19]. Furthermore, Industry 4.0 has also led to the creation of new standards for developing industrial automation systems. IEC 61499 [IEC12] is an international standard that has been gaining popularity recently due to its capability to develop complex and distributed systems [Vya11]. It is the successor of IEC 61131-3 [IEC02], which was the state-of-the-art standard for developing automation systems.

Although the fourth industrial revolution seems promising for the future of automation systems, new challenges arise. First of all, in order to fully take advantage of automation, developers must also consider external elements associated with the system. In particular, modern automation systems are equipped with many sensors and actuators that interact with nondeterministic environments. Taking into account the probabilistic behaviours originating from this interaction can help optimise the system's quantitative aspects, such as productivity. Secondly, complex systems that incorporate the aforementioned technologies are costly to develop. Therefore, these systems are expected to last for a long time and must be able to evolve when some new requirements arrive after the deployment phase. Designing the target system that can satisfy these requirements is complicated. Manual modifications may introduce erroneous and unexpected behaviours to the system.

1.1 Context

Industrial automation systems consist of mechanical components, sensors, actuators, control devices, and control applications [MR14]. Mechanical components are physical parts of the system (e.g., a conveyor). Sensors read the environment state (e.g., the temperature) and send signals to the control devices, whereas actuators receive signals from the control devices and activate the mechanical components (e.g., start the conveyor). Control devices, such as Programmable Logic Controller [Ley92], are the systems' processing units. This thesis focuses on the control applications that reside in these devices. Each application defines the behaviour of the automation system using certain programming standards.

IEC 61499 [IEC12] is a promising industrial automation standard in the era of Industry 4.0. The standard enables application-centric design, allowing developers to focus on designing the system's behaviour at the application level. Once this step is completed, the software components can be freely distributed to control devices in the system. Furthermore, the standard is notable for its reusability, reconfigurability, interoperability, and portability [Vya11; Pan+14]. More precisely, the building block of an application, called Function Block (FB), can be reused for different applications without integration issues. Reconfiguration of FBs and their respective control devices can be done at runtime without stopping the execution, allowing downtimeless evolution of the system [Hum+06; SVZ13]. Interoperability means that applications can execute on devices from any vendor. Portability is supported because the development of applications is not restricted to specific software tools. Moreover, the standard is favoured by researchers in the industrial automation domain who continuously propose new verification and evolution techniques because of its potential. There have been over 300 scientific articles on IEC 61499 published in the past 20 years [1]. Ultimately, the methodology proposed in this thesis is aimed at industrial automation systems that are based on the IEC 61499 standard.

Formal methods apply various mathematical techniques in computer science for the design and verification of software and hardware systems [ORe17; GBP20]. Their application in the industrial automation domain has been increasing in recent years [Sin+19] due to their robustness and capability to mathematically ensure the system's correctness and identify erroneous behaviours. Furthermore, the continuous development of verification tools makes them more accessible for practitioners in the industry [Alm+11]. For these reasons, this thesis employs formal methods to support long-running and evolving industrial automation systems.

[1] source: `https://dblp.org/search/publ?q=IEC+61499`

Requirements are conditions specified by the user to be possessed by the system for achieving specific objectives [ANS83]. The taxonomy described in [Gli07] classifies requirements into functional requirements, performance requirements, quality requirements, and constraints [2]. A functional requirement is associated with the system's behaviour. Performance requirements specify measurable aspects. Quality requirements define the system's characteristics. Constraints are a set of restrictions the system must comply with. This thesis focuses on functional requirements, specifically those involving the system's behaviours that can be associated with sequences of executions at the application level.

1.2 Motivations

Complex IEC 61499 applications consist of many FBs distributed onto multiple control devices associated with many sensors and actuators [Par+23]. Such applications exhibit unpredictable behaviours due to their interactions with the environment, which involves various external elements that may impact the system's behaviour. For example, the performance of a power plant can be influenced by the surrounding temperature [DA11]. A recent study shows that warm conditions prevent the plant from generating optimal power [Sha+24]. Another example is that workers in a manufacturing factory may have different methods or strategies when interacting with the system's mechanical components. Some workers may work in a certain way that can be inefficient and thus impact productivity. Overall, these external elements are often unpredictable, but it is important to analyse and understand them because they can impact the system's quantitative aspects. A helpful approach is to use techniques that can analyse the likelihood of nondeterministic events associated with the system's external elements.

Formal verification techniques, such as model checking [BK08], have been proposed in [OV21; Dro+16; BRS17] to ascertain the correctness of IEC 61499 systems. However, static techniques are insufficient to take into account the probabilistic behaviours originating from the environment. This is because they only consider application behaviour at design time, whereas the likelihood of executions can only be observed at runtime. Altogether, the existence of the environment in industrial automation systems motivates the following research question.

[2] The last three classes are often categorised as non-functional requirements.

RQ1: How to formally verify and analyse the quantitative aspects of IEC 61499 applications by considering their probabilistic behaviours originating from the environment?

Industrial automation systems undergo evolutions during their lifecycle to satisfy certain requirements. IEC 61499 standard provides downtimeless evolution frameworks [SVZ13] that are useful in this context. However, designing the target application that satisfies the requirements is a tedious and time-consuming task. This task becomes even more challenging when there are many possible modifications, such as changing the FBs' internal functionalities and creating new connections between FBs. Manual efforts may result in unsatisfied requirements, unexpected behaviours, and extra complexity.

Existing works [PS21; PHS22a; PHS22b; WSZ20; Son+21; Son+22] mostly focus on facilitating the reconfiguration process (i.e., the steps required to safely modify the application at runtime). They do not define how to obtain the target application systematically. Hence, the user must provide it as input. Consequently, there is no notion of requirements that should be satisfied after the application has been modified. Ultimately, there is a need for techniques that can automatically compute the target application of an initial IEC 61499 application to satisfy certain requirements. Such techniques have not yet been proposed to the best of our knowledge. Therefore, this thesis also focuses on the following research question.

RQ2: How to automatically compute IEC 61499 applications that satisfy some given requirements?

1.3 Approach

For the first research question, *probabilistic model checking* is proposed to formally verify and analyse the quantitative aspects of IEC 61499 applications. *Runtime enforcement* and *guided evolution* are proposed for the second research question to generate new IEC 61499 applications from given initial applications and input requirements.

Probabilistic Model Checking (PMC) [KNP18] is a formal method to verify the quantitative properties of stochastic systems. It enables users to check the likelihood of certain execution sequences, which helps verify the system's probabilistic behaviours originating from the environment. For instance, when a certain temperature interval is desirable for improving the system's performance, PMC can check its likelihood by

checking the probability of the corresponding execution sequences. PMC is applied to an IEC 61499 application by first creating the formal model of its behaviour. This model consists of states and transitions labelled with events and data that the application can trigger. Next, a monitor is created and integrated into the application to record execution traces, which are sequences of events and data. Probabilistic models are then computed by enriching the transitions of the initial model with the probabilistic values computed from the execution trace. Finally, these enriched models can be verified using the probabilistic model checker for some given probabilistic properties.

Runtime enforcement [Fal10; Fal+18] is a method to ensure system correctness during its execution. The idea is to integrate an enforcer, synthesised from a given property, to modify the application execution whenever the property is violated. This concept is adopted for evolving IEC 61499 applications according to the requirements. A lightweight and expressive language, called contract automata, is proposed to specify the requirements. A contract automaton specified by the user is synthesised into an enforcer in the form of an FB. This enforcer FB is integrated into the application so that it executes according to the automaton. The approach simplifies the evolution process by automatically adding an enforcer to modify the execution instead of manually modifying the FBs and their connections to make the application execute as specified in the given requirements.

As an alternative to runtime enforcement, guided evolution techniques for IEC 61499 applications are proposed. This approach takes as input an initial application and some given requirements to generate guidelines for obtaining the target application. The approach relies on the application behavioural model. The main idea is to modify parts of the model to make them comply with the requirements. Then, the initial and modified models are compared to generate evolution guidelines. These guidelines can bridge the gap between the requirements and the target application. Traditionally, developers may need to consider many options when modifying the application according to the requirements. The approach helps these developers to determine the target application by considering the generated guidelines.

1.4 Contribution

This thesis presents four contributions. Each of them is demonstrated with a unique IEC 61499 application as a case study because the proposed methods are better

explained using specific cases. For example, the probabilistic model checking approach is demonstrated with a drilling station. This system is suitable because it has a nondeterministic aspect originating from the operators who place and take the industrial materials on the rotating table.

The contributions of this thesis are summarised as follows:

Contribution 1: Formal modelling of IEC 61499 applications
A method to automatically generate behavioural models of IEC 61499 applications is proposed. An application is first translated into a specification language called LNT (LOTOS New Technology) [Cha+24]. Afterwards, the CADP toolbox [Gar+13] is used to compile the specification into a Labelled Transition System (LTS) behavioural model [Kel76]. The modelling approach is applied on an IEC 61499 blinking application. The generated model captures the sequences of events and data that the application can trigger.

Contribution 2: Probabilistic model checking of IEC 61499 applications
Probabilistic model checking is applied to verify and analyse the quantitative aspects of IEC 61499 applications. The approach consists of monitoring techniques and probabilistic model computation. An application is monitored by adding an FB dedicated to recording the execution traces. A trace and an initial LTS model are then used to compute a Probabilistic Transition System (PTS), which is an LTS extended with probabilistic values on its transitions. A tool to automate this computation is developed. The approach is demonstrated using an IEC 61499 drilling station for manufacturing industrial materials. Several properties are checked on the generated PTS models (computed from different traces) using the CADP probabilistic model checker. The results are helpful for analysing the impact of the environment with respect to the system's quantitative aspects, such as productivity.

Contribution 3: Runtime enforcement for IEC 61499 applications
Runtime enforcement is applied to make IEC 61499 applications satisfy some given requirements. This contribution consists of (i) a contract automata specification language and (ii) enforcer synthesis and integration methods. Contract automata is the input language for specifying the evolution requirements. A contract automaton is a finite-state machine in which its transitions can be typed to specify the modification of events and data. This automaton is synthesised into an enforcer in the form of a basic FB, which is composed of interfaces and an Execution Control Chart (ECC). An enforcer is then integrated by modifying the set of connections in the application. The approach satisfies *soundness* and *transparency* characteristics, whereas *deadlock-freedom* can be verified with model checking. An IEC 61499 conveyor test station is used as a case study to present the runtime enforcement approach. A tool

to automate the synthesis of enforcers is developed, and experiments to show that enforcers do not induce performance overheads are conducted.

Contribution 4: Guided evolution of IEC 61499 applications
An approach to generate guidelines for evolving IEC 61499 applications is proposed. The approach consists of three modules: (i) preliminary, (ii) model evolution, and (iii) guidelines generation. The preliminary module extends the modelling techniques to generate behavioural models that involve both the inputs and outputs of FBs in the application. The model evolution module identifies relevant parts of the model and modifies them according to the input requirements. The guidelines generation compares the initial and modified models to generate the steps required to modify the application. The guided evolution approach is demonstrated using an IEC 61499 mixing tank system as a case study.

1.5 List of publications

The aforementioned contributions resulted in the following publications:

- Yliès Falcone, Irman Faqrizal, Gwen Salaün. *Runtime Enforcement for IEC 61499 Applications*. In: Proc. of SEFM'22, Vol. 1355, LNCS, Springer, 2022, pp. 352-368. (Contribution 3)
- Yliès Falcone, Irman Faqrizal, Gwen Salaün. *Probabilistic Analysis of Industrial IoT Applications*. In: Proc. of IoT'22, ACM, 2022, pp. 41-48. (Contributions 1 and 2)
- Irman Faqrizal, Tatiana Liakh, Midhun Xavier, Gwen Salaün, Valeriy Vyatkin. *Probabilistic Model Checking for IEC 61499: A Manufacturing Application*. In: Proc. of ICIT'24, IEEE, 2024, pp. 1-6. (Contributions 1 and 2)
- Irman Faqrizal, Gwen Salaün, Yliès Falcone. *Guided Evolution of IEC 61499 Applications*. In: Proc. of ETFA'24, IEEE, 2024. (Contribution 4)
- Irman Faqrizal, Gwen Salaün, Yliès Falcone. *Adaptive Industrial Control Systems via IEC 61499 and Runtime Enforcement*. In: ACM Trans. Autonom. Adapt. Syst, 2024. (Contribution 3)

1.6 Outline

The rest of this thesis is structured as follows:

Chapter 2: Preliminaries
This chapter introduces relevant background notions. It begins with the notations

used in the rest of the thesis. The IEC 61499 standard is then explained. Afterwards, the notion of formal methods is discussed, in particular, probabilistic model checking and runtime enforcement techniques.

Chapter 3: Formal modelling of IEC 61499 applications
This chapter explains how to generate behavioural models of IEC 61499 applications. The case study for this chapter is first described. Then, the definition of an IEC 61499 behavioural model is given. Afterwards, the methods to translate IEC 61499 applications into LNT specifications and to generate LTS models are explained.

Chapter 4: Probabilistic model checking of IEC 61499 applications
This chapter presents an approach to apply probabilistic model checking to IEC 61499 applications. A drilling station manufacturing system is first introduced as a case study. Afterwards, the monitoring techniques and the probabilistic model computation are explained. Next, the tool support to automate the approach is presented. Experimental results are then presented to demonstrate that the approach can be used to analyse the quantitative aspects of the drilling station.

Chapter 5: Runtime enforcement for IEC 61499 applications
This chapter presents the runtime enforcement techniques for IEC 61499 applications. It starts with a description of a conveyor test station as a case study to illustrate the approach. The IEC 61499 runtime enforcement problem is then defined. Afterwards, a specification language called contract automata is proposed. Next, the synthesis and integration of enforcers are explained. Runtime enforcement characteristics are then explained. Finally, implementation and experiments are presented.

Chapter 6: Guided evolution of IEC 61499 applications
This chapter presents a method to guide the evolution of IEC 61499 applications. It begins with a section on an extended version of the behavioural model. Then, a mixing tank system is described as a case study. Afterwards, the three modules for generating evolution guidelines are explained. Finally, a prototype of the approach's implementation is shown.

Chapter 7: Related work
This chapter surveys related works and compares them with the thesis's contributions. Firstly, LTS and PTS models are compared with existing IEC 61499 formal models. Then, several verification techniques for the standard are surveyed. Finally, the differences between our evolution methods and the existing ones are explained.

Chapter 8: Conclusion
This chapter concludes the thesis with a summary of the works and possible perspectives to be considered in the future.

2

Preliminaries

This chapter introduces relevant background notions. Section 2.1 introduces some notations used in this thesis. Section 2.2 describes the IEC 61499 standard. Section 2.3 explains probabilistic model checking and runtime enforcement methods. Section 2.4 concludes the chapter.

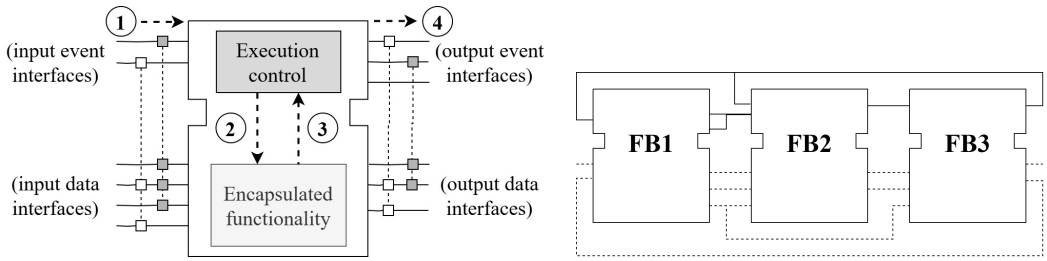
2.1 Notations

The following notations are used in the rest of this thesis:

- $\mathbb{S}$ is the set of strings.
- $\mathbb{B}$ is the set of boolean values.
- $\mathbb{Z}$ is the set of integers.
- $\mathbb{N}$ is the set of natural numbers.
- $\mathbb{V}$ is the set of variables. A variable $v = (x, y) \in \mathbb{V}$ consists of a name $x \in \mathbb{S}$ and a value $y \in \mathbb{B} \cup \mathbb{Z}$.
- B_V represents a boolean expression over a set of variables $V \subseteq \mathbb{V}$. For instance, B_V is an expression $(a = true) \wedge (b \neq null)$ over $V = \{(a, false), (b, 3)\}$. We can also write this expression as $a = true$ or a.
- $\mathbb{G}$ is the set of bags. A *bag* can be used to store any type of element, allowing multiple occurrences. For instance, $\mathbb{G}_\mathbb{Z}$ is the set of bags over integers; each bag $b_\mathbb{Z} \in \mathbb{G}_\mathbb{Z}$ can be used to store integers (e.g., $b_\mathbb{Z} = \{1, 1, 2, 3\}$),
- ϵ represents an empty value.

2.2 IEC 61499

IEC 61499 [IEC12] is a standard for designing industrial automation systems. It is the successor of IEC 61131-3 [IEC02]. Unlike its predecessor, which uses a cyclic execution model, IEC 61499 adopts an event-driven execution model. The standard defines the behaviour of an application using interconnected function blocks. A



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

Figure 2.1: FB and IEC 61499 application

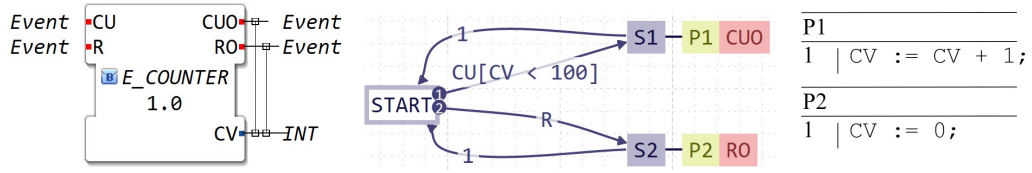
Function Block (FB) is connected to other FBs through its input and output event and data interfaces (see Figure 2.1). An FB encapsulates an internal behaviour that is executed when one of its input event interfaces receives an event. IEC 61499 applications presented in this thesis are designed using 4diac IDE [Str+08].

The execution model of an FB in IEC 61499 standard [ZL14], depicted in Figure 2.1a, is as follows: (1) an event is received, and the values of the associated data interfaces are updated, (2) based on the FB's current state in the execution control, an encapsulated functionality is executed, (3) this execution assigns new values to the output data interfaces, and the execution control determines the output event to be sent, (4) the output event is sent, and the values of the associated output data interfaces are updated. This is the generic execution model. In practice, steps 2 to 4 can be repeated multiple times depending on the FB's type and the execution control. Furthermore, a single activation of the FB can be defined as the period between the beginning of step 1 and the termination of step 4. Once an input event activates the FB, another event cannot enter before the previous activation has finished.

An IEC 61499 application is illustrated in Figure 2.1b. For clarity, connections between event and data interfaces are represented with different styles: solid lines for event connections and dashed lines for data connections. The standard allows both *fan-in* (e.g., $FB1$ to $FB2$) and *fan-out* ($FB3$ to $FB1$ and $FB2$) connections for event interfaces. However, it only allows *fan-out* (e.g., $FB1$ to $FB2$ and $FB3$) for data interface connections.

2.2.1 Types of function block

There are three types of FB: Basic FB (BFB), Service Interface FB (SIFB), and Composite FB (CFB). A BFB defines its behaviour using a Moore-type finite state machine called the Execution Control Chart (ECC). SIFB behaviours are specific to their control device. A CFB is composed of a network of FBs. Note that when



(a) FB and its interfaces **(b)** ECC **(c)** ST programs

Figure 2.2: Example of BFB

referring to an FB in an application, we use the name of the instance instead of the FB's name (e.g., $CY1$ instead of E_CYCLE in Figure 2.3) because there can be multiple FBs with the same name.

Basic Function Block. In a BFB, the aforementioned execution control is specified using ECC, and encapsulated functionality is expressed as programs written in Structured Text [Ant20] [1]. In Figure 2.2, an example of a basic FB named $E\_COUNTER$ [2] with its ECC and ST programs is presented. This FB behaves like a counter. It counts how many times CU receives an event and shows the result on CV. An example of this FB activation is as follows. When an event arrives at CU, the FB is activated. First, it checks the current state of the ECC. By design, the current state of this FB at the beginning of an activation is always $START$. From this state, if CV is less than 100, then $S1$ is visited, and $P1$ is executed to increment CV by one. Afterwards, an output event CUO is sent. The transition labelled with 1 (i.e., $true$) is automatically traversed to come back to $START$, and then the activation terminates. When R receives an event, the transition from $START$ to $S2$ is traversed without checking the value of CV. Then, $P2$ sets the value of CV to 0, and event RO is sent. A more detailed explanation of the basic FB's execution model, including the ECC's Operation State Machine (OSM), is given in [Dro+21].

Service Interface Function Block. SIFBs give access to the services provided by the control device where the IEC 61499 application resides. Device manufacturers have unique methods for creating SIFBs. Therefore, unlike BFBs, their internal behaviours are often hidden. The most common service is access to the sensors and actuators in the system. SIFBs that can read sensor states are called sensor SIFBs, whereas the ones that can set actuator states are called actuator SIFBs.

In Figure 2.3, $S1$ is a sensor SIFB that sends an event from CNF with $IN = true$ if the current state of the sensor reads positive; otherwise, it sends an event from CNF with $IN = false$. A sensor SIFB is coupled with an FB that can send an event

[1] ST programs are often called algorithms in the literature. The term ST program is used to distinguish algorithms written in ST from algorithms proposed in this thesis.
[2] This is a simplified version of an FB from the standard library called E_CTU.

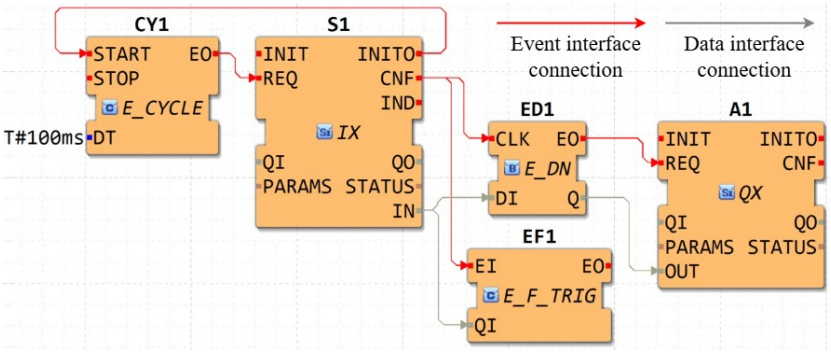


Figure 2.3: Example of IEC 61499 application with SIFBs

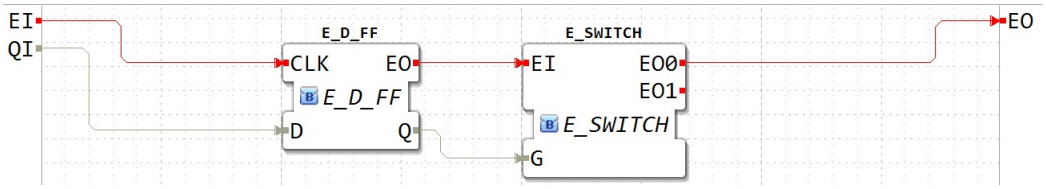


Figure 2.4: Network of FBs inside E_F_TRIG

periodically, such as $CY1$, to keep reading the sensor's state. $A1$ is an actuator SIFB, which controls the actuator's state. Suppose the actuator is the motor of a conveyor. Then, the conveyor starts (or stops) when REQ receives an event with $OUT = true$ (or $OUT = false$).

Composite Function Block. CFBs are used mostly to design applications where the system can be divided into several subsystems [Son+21]. A subsystem consisting of several FBs is represented with a single CFB. A CFB can also be a combination of several BFBs to serve a specific purpose. For instance, $EF1$ in Figure 2.3 is an instance of E_F_TRIG, which is a CFB composed of two BFBs shown in Figure 2.4. Applications with CFBs can always be flattened [DV08]. Such applications only consist of BFBs and SIFBs.

2.2.2 Syntax

This section explains the syntax of IEC 61499. We focus on describing the components of a BFB because the SIFBs' execution controls and encapsulated functionalities are hidden. Also, applications with CFBs can be flattened so that they consist of only BFBs and SIFBs.

Interfaces. FB's interfaces consist of event and data input and output interfaces. An event interface is a string, whereas a data interface is a variable that can store a

value. An event interface and a data interface can be associated to update the data interface value when the associated event interface receives or sends an event.

Definition 2.2.1 *(Interfaces)* Interfaces are a tuple (EI, EO, DI, DO, AI, AO), where:

- $EI = \{ei_1, ei_2, ..., ei_n \in \mathbb{S}\}$ is a set of input event interfaces,
- $EO = \{eo_1, eo_2, ..., eo_n \in \mathbb{S}\}$ is a set of output event interfaces,
- $DI = \{di_1, di_2, ..., di_n \in \mathbb{V}\}$ is a set of input data interfaces,
- $DO = \{do_1, do_2, ..., do_n \in \mathbb{V}\}$ is a set of output data interfaces,
- $AI \subseteq EI \times DI$ is a set of input event and data interface associations,
- $AO \subseteq EO \times DO$ is a set of output event and data interface associations.

We use function $w : EI \cup EO \rightarrow 2^{DI} \cup 2^{DO}$ to obtain the set of associated data interfaces for a given event interface.

Internal data. In addition to DI and DO, data can also be stored internally. The internal data in a BFB is denoted by $DV = \{dv_1, dv_2, ..., dv_n \in \mathbb{V}\}$. These data can be part of the BFB's ECC, but their value cannot be shared with other FBs in the application. The set of all data in a BFB is denoted by $DA = DI \cup DO \cup DV$.

ST programs. The modification of data values in a BFB is specified using Structured Text (ST) [Ant20]. We use a fragment of the ST syntax shown in (2.1), where $x \in \mathbb{S}$ is a variable name, $y \in \mathbb{B}$ and $z \in \mathbb{Z}$ are boolean and integer values.

$$p ::= p_1 \ . \ p_2 \mid x_1 := x_2 \ ; \mid x := y \ ; \mid x_1 := x_2 + z \ ; \mid x_1 := x_2 - z \ ; \tag{2.1}$$

The syntax shows that each program is a sequence of assignments, additions, and subtractions. ST programs written with this syntax are denoted by $\mathbb{P}$. Furthermore, programs are written inside $\langle \ \rangle$ brackets to distinguish them from other notations.

Execution control chart. The ECC determines a BFB's output events and data every time one of its input event interfaces is triggered. An ECC is composed of states and transitions. A state consists of a sequence of operations, whereas a transition is labelled with a guard. An operation $r = (p, eo)$ is composed of a program $p \in \mathbb{P}$ and an output event interface $eo \in EO$.

A guard is written as $[\![ei[B_{DA}]]\!]$, where $ei \in EI$ is an input event interface and B_{DA} is a boolean expression over the set of all data DA. A guard with empty expression $[\![ei[true]]\!]$ (or empty event interface $[\![\epsilon[B_{DA}]]\!]$) is written $[\![ei]\!]$ (or $[\![[B_{DA}]]\!]$). An empty guard (i.e., $[\![\epsilon[true]]\!]$) is written as 1. A transition labelled by an empty guard is called an unguarded transition.

Definition 2.2.2 *(Execution control chart)* An ECC is a tuple (Se, se^0, Ge, Te), where:

- Se is a set of states, and each state $se \in Se$ consists of a sequence of operations $r_1, r_2, ..., r_n$,
- $se^0 \in Se$ is the initial state,
- Ge is a set of guards,
- $Te \subseteq Se \times Ge \times Se$ is a set of transitions.

Basic function block. We can now define a BFB as an FB that consists of interfaces, internal data, programs, and ECC. SIFBs use the same notation as BFBs, but the last three components are empty.

Definition 2.2.3 *(Basic function block)* A BFB is a tuple (itf, DV, P, ecc), where:

- $itf = (EI, EO, DI, DO, AI, AO)$ is the interfaces,
- $DV = \{dv_1, dv_2, ..., dv_n \in \mathbb{V}\}$ is a set of internal data,
- $P = \{p_1, p_2, ..., p_n \in \mathbb{P}\}$ is a set of programs,
- $ecc = (Se, se^0, Ge, Te)$ is the ECC.

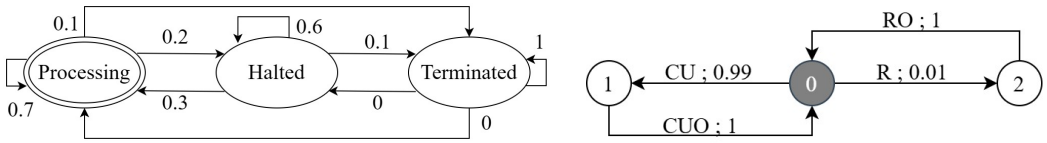
Application. Finally, an IEC 61499 application consists of a set of FBs and connections between their event and data interfaces.

Definition 2.2.4 *(Application)* An application is a tuple (FB, EC, DC), where:

- $FB = \{fb_1, fb_2, ..., fb_n\}$ is a set of FBs,
- $EC \subseteq EO' \times EI'$ is a set of event connections, $EO' = EO_1 \cup EO_2 \cup ... \cup EO_n$ and $EI' = EI_1 \cup EI_2 \cup ... \cup EI_n$ are the event interfaces of $fb_1, fb_2, ..., fb_n$,
- $DC \subseteq DO' \times DI'$ is a set of data connections, $DO' = DO_1 \cup DO_2 \cup ... \cup DO_n$ and $DI' = DI_1 \cup DI_2 \cup ... \cup DI_n$ are the data interfaces of $fb_1, fb_2, ..., fb_n$.

2.3 Formal methods

Formal methods [ORe17; GBP20] include various mathematical techniques to support system design, including specification, analysis, and verification. These techniques are helpful in ensuring the dependability and reliability of industrial automation systems [Sin+19]. Specific types of formal methods need to be selected according to the objectives of this thesis. The first objective is to verify and analyze the quantitative aspects of the system. For this, a verification method that considers the likelihood of executions is required. Hence, probabilistic model checking is chosen. The second objective is to evolve the application according to some given



(a) Example of DTMC **(b)** Example of PTS

Figure 2.5: Probabilistic models

requirements, for which a method that can modify the runtime execution is required. Therefore, runtime enforcement is chosen.

2.3.1 Probabilistic model checking

Model checking [BK08] is a method that verifies whether a system model satisfies a given property. The model checker returns a counterexample, a sequence of actions leading to property violation when the property is unsatisfied. This method can assert properties such as something bad never happens (i.e., safety) or something good eventually happens (i.e., liveness).

Probabilistic Model Checking (PMC) can be regarded as an extension of model checking. A system is modelled as a probabilistic model to be verified with probabilistic properties. Beyond verifying the system's correctness according to the properties, PMC allows one to verify the likelihood of those properties being satisfied. Discrete-Time Markov Chain (DTMC) [KNP18] is a commonly used probabilistic model formalism. Figure 2.5a shows an example of a DTMC that models three possible states of a particular system with the probabilities of moving from one state to another. An example of a probabilistic property in the PRISM [KNP02] language for this model is $P = 1\ [F\ \text{“}Terminated\text{”}]$, which states that the probability of the system termination must be equal to 1.

In this work, we use the CADP [Gar+13] probabilistic model checker. A system is specified in the LNT (LOTOS New Technology) specification language [Cha+24], an extension of LOTOS [ISO89], an ISO-standardized process algebra. We present a fragment of LNT syntax in Table 2.1. In this table, π is a process name, e is an event, d is a variable name, k is a type, and y is a boolean expression. The syntax consists of the process definition (**`process`**), sequential execution (;), assignment ($:=$), process call ($\pi[\,](\,)$), select statement (**`select`**), hiding of actions (**`hide`**), conditional statement (**`if`**), variable definition (**`var`**), and parallel composition (**`par`**). An action can be defined as an internal action (i). It can also be defined as an event representing an input (?) or output (!) communication gate for data. An LNT

Table 2.1: Fragment of LNT

$l ::=$	**process** $\pi[e_0, e_1, ... :$ **any**$](d_0 : k, d_1 : k, ...)$ **is** x **end process**
$x ::=$	$a \mid x_1 \ ; \ x_2 \mid d_0 := d_1 \mid \pi[e_0, e_1, ...](d_0, d_1, ...)$
	$\mid$ **select** $x_1[\,]...[\,]x_n$ **end select** $\mid$ **hide** $d : k$ **in** x_1 **end hide**
	$\mid$ **if** (y) **then** x_1 **else** x_2 **end if** $\mid$ **var** $d : k$ **in** x_1 **end var**
	$\mid$ **par** $\{e_{(1,1)}, e_{(2,1)}, ...\} \rightarrow x_1 \| ... \| \{e_{(1,n)}, e_{(2,n)}, ...\} \rightarrow x_n$ **end par**
$a ::=$	i $\mid e \mid e(?\,d_0, ?\,d_1, ..) \mid e(!\,d_0, !\,d_1, ..)$

Table 2.2: Fragment of MCL5

$f ::=$	$a \mid$ **true** $\mid$ **false** $\mid$ **not** $a \mid a_1$ **or** $a_2 \mid a_1$ **and** a_2
$r ::=$	$f \mid$ **nil** $\mid r_1$ **.** $r_2 \mid r_1 \mid r_2 \mid r$*****
$m ::=$	**prob** r **is** $o\ p$ **end prob**

specification can be compiled into a Labelled Transition System (LTS) behavioural model [Kel76], which is a state machine consisting of states and transitions labelled with events.

CADP supports PMC by extending LTS into Probabilistic Transition Systems (PTSs), where transitions are also labelled with probabilities. Figure 2.5b presents an example of a PTS model of the $E\_COUNTER$ FB in Figure 2.2 [3]. The initial state is highlighted to distinguish it from other states. The PTS describes the probabilities of event sequences. For example, the probability of CU followed by CUO and then RO is 0.0099. Both DTMC and PTS can represent industrial automation systems as probabilistic models. However, PTS is chosen because IEC 61499 adopts an event-driven execution model. The transition probability represents the likelihood of executing an event rather than the likelihood of being in a certain state. PTS facilitates system verification at the application level by specifying the probability of sequences of events and data that the application can trigger.

In CADP, probabilistic properties are written in MCL [MR18]. We use a fragment of the MCL syntax shown in Table 2.2. In this fragment, a is an action, **true, false, not, or, and** are logical operators, **nil** is an empty operator, **.** is a concatenation operator, $\mid$ is a choice operator, ***** is a transitive and reflexive closure operator, o is a comparison operator (i.e., $>, <, >=, <=, =, <>$), and $p \in [0, 1]$ is a probability. The symbol $?$ can be added before p to return the probability of sequences of executions described by r. An example of a probabilistic property for

[3] The methods for generating LTSs and PTSs are explained in the following chapters.

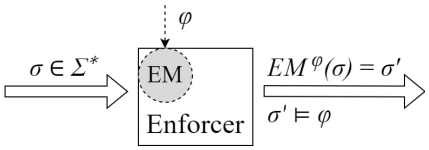


Figure 2.6: Conceptual view of runtime enforcement

the PTS in Figure 2.5b is **prob** R . RO **is** > 0 **end prob**, which states that the probability of resetting the counter must be greater than 0.

2.3.2 Runtime enforcement

Runtime enforcement [Fal10; Fal+18] is a method for enforcing certain requirements to the application's behaviour by modifying its runtime execution. This modification is instrumented by an additional component named *enforcer*.

Runtime enforcement is illustrated in Figure 2.6. An enforcer receives as input a trace $\sigma \in \Sigma^*$, a sequence of executions produced by the system. Σ^* represents the set of all possible sequences of executions. The enforcer encapsulates an Enforcement Mechanism (EM), which determines the modification of the trace. The enforcer and its EM are synthesised from requirements specified as φ. EM is defined as a function $EM^{\varphi} : \Sigma^* \rightarrow \Sigma^*$, where for a given input trace σ, it returns an output trace σ' that satisfies the requirements φ, i.e., $EM^{\varphi}(\sigma) = \sigma'$, such that $\sigma' \vDash \varphi$.

Let us illustrate runtime enforcement with an example within the context of IEC 61499. Suppose a trace of the $E\_COUNTER$ in Figure 2.2 is defined as the sequence of events triggered from the output event interfaces. For instance, it produces a trace $\sigma = CUO, CUO, RO, ...$ Requirements can be formulated using LTL (Linear Temporal Logic) [Pnu77], such as $\varphi = \Box CUO$. It means that the application should only generate sequences of CUO. Let us assume that EM replaces every RO with CUO. Therefore, $EM^{\varphi}(\sigma) = \sigma' = CUO, CUO, CUO, ...$ This example is given only to illustrate how runtime enforcement works in practice. Chapter 5 discusses the application of runtime enforcement to IEC 61499 in more detail.

2.4 Concluding remarks

This chapter has presented some preliminary concepts, including the notations used in the thesis, the IEC 61499 standard, probabilistic model checking, and runtime enforcement.

3

Formal modelling of IEC 61499 applications

This chapter describes an approach to formally model the behaviour of IEC 61499 applications. A model of a system is an abstract representation of the system. There are many different ways to design a system's model because different levels of abstraction exist [AS96]. This thesis focuses on modelling the system's behaviour at the application level, which aligns with the type of input requirements we aim for (as explained in Chapter 1). An IEC 61499 application is modelled as a Labelled Transition System (LTS). Transitions in an LTS are labelled with events and data that the FBs in the application can trigger. The formal modelling approach limits data type to boolean to avoid state space explosion. This limitation is not an issue since boolean values are sufficient to define the control logic of industrial systems relying on IEC 61499.

The overview of the formal modelling approach is shown in Figure 3.1. Each FB in the application is first translated into an LNT process. Afterwards, a main process consisting of a parallel composition of the translated processes is created. Finally, the LNT specification is compiled into an LTS model.

This chapter is organised as follows. Section 3.1 presents a case study. Section 3.2 defines the IEC 61499 behavioural model. Section 3.3 explains the translation from IEC 61499 FBs to LNT processes. Section 3.4 describes how LNT main processes are built and compiled into LTSs. Section 3.5 concludes.

3.1 Case study

The formal modelling techniques introduced in this chapter are illustrated using an IEC 61499 blinking application [IDE24] in Figure 3.2. The application consists of three FBs. DL triggers an event periodically from EO. As described in its ECC (Figure 3.2b), SW sends an event from $EO0$ if $G = \mathit{false}$; otherwise, it sends $EO1$.

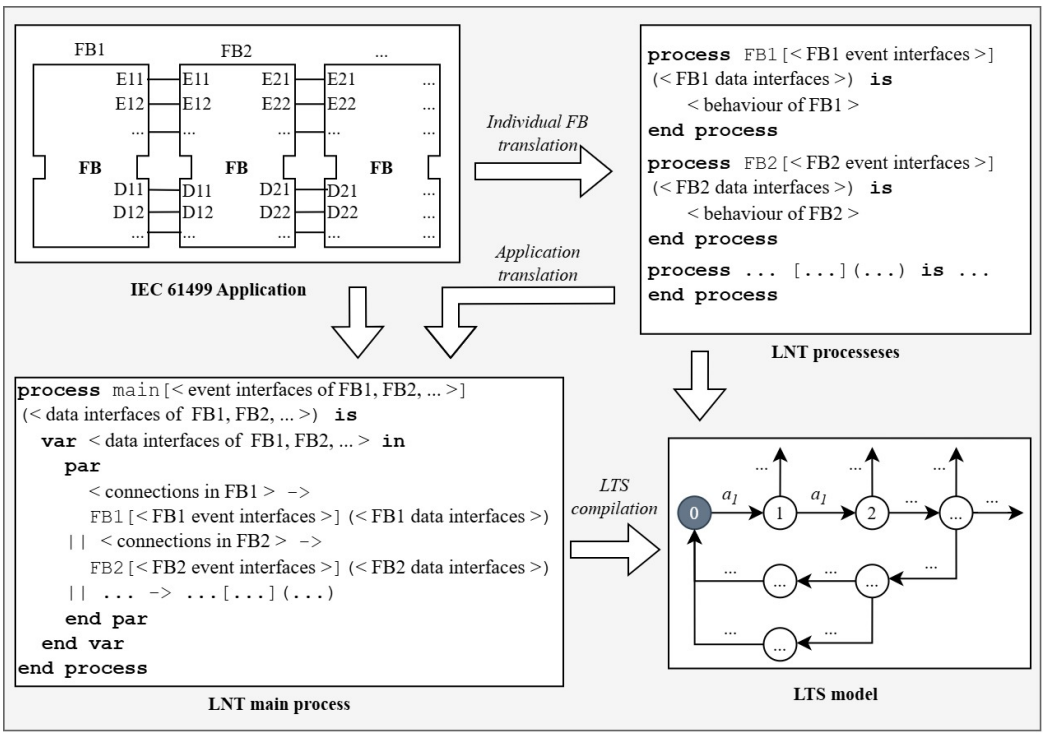
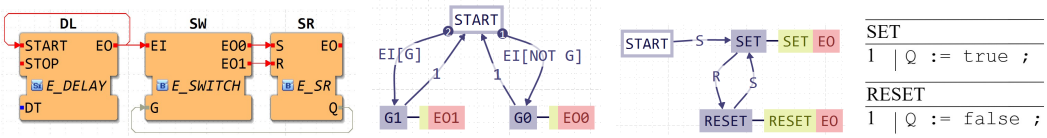


Figure 3.1: Overview of the formal modelling techniques



(a) Application **(b)** *SW* ECC **(c)** *SR* ECC **(d)** ST programs

Figure 3.2: IEC 61499 blinking application

Finally, the value of Q in SR changes to $true$ (or $false$) if S (or R) receives an event. The network of FBs results in an output data interface, Q, whose value alternates between $true$ and $false$ (thus known as the blinking application). The syntactic components of the blinking application are presented in Table 3.1.

3.2 IEC 61499 behavioural model

In this section, the notion of the IEC 61499 behavioural model is first defined. As a first step, an atomic execution in the application is defined. This execution is called an action, which consists of an event and data generated by an FB.

Definition 3.2.1 (Action) *An action is a tuple* (fb, e, D)*, where:*

Table 3.1: Components of the IEC 61499 blinking application

iec	(FB, EC, DC)
FB	$\{DL = (itf_1, \epsilon, \epsilon, \epsilon), SW = (itf_2, \varnothing, \varnothing, ecc_2), SR = (itf_3, \varnothing, P_3, ecc_3)\}$
itf_1	$\big(EI_1 = \{START, STOP\}, EO_1 = \{EO\}, DI_1 = \{(DT, s)\}, DO_1 = \varnothing,$ $WI_1 = \{(START, DT)\}, WO_1 = \varnothing\big)$
itf_2	$\big(EI_2 = \{EI\}, EO_2 = \{EO0, EO1\}, DI_2 = \{G\}, DO_2 = \varnothing, WI_2 = \{(EI, G)\}, WO_2 = \varnothing\big)$
itf_3	$\big(EI_3 = \{S, R\}, EO_3 = \{EO\}, DI_3 = \varnothing, DO_3 = \{Q\}, WI_3 = \varnothing, WO_3 = \{(EO, Q)\}\big)$
P_3	$(SET = \langle$`Q := true ;`$\rangle, RESET = \langle$`Q := false ;`$\rangle)$
ecc_2	$\big(Se_2 = \{START, G0 = (\epsilon, EO0), G1 = (\epsilon, EO1)\}, se_2^0 = START,$ $Ge_2 = \{[\![EI[G]]\!], [\![EI[NOT\ G]]\!]\}, Te_2 = \{(START, [\![EI[NOT\ G]]\!], GO),$ $(GO, 1, START), (START, [\![EI[G]]\!], G1), (G1, 1, START)\}\big)$
ecc_3	$\big(Se_3 = \{START, SET = (SET, EO), RESET = (RESET, EO)\}, se_3^0 = START,$ $Ge_3 = \{[\![R]\!], [\![S]\!]\}, Te_3 = \{(START, [\![S]\!], SET), (SET, [\![R]\!], RESET),$ $(RESET, [\![S]\!], SET)\}\big)$
EC'	$\{(EO, START), (EO, EI), (EO0, S), (EO1, R)\}$
DC'	$\{(Q, G)\}$

- *$fb \in \mathbb{S}$ is an FB,*
- *$e \in \mathbb{S}$ is an output event,*
- *$D = \{d_1, d_2, ..., d_n \in \mathbb{V}\}$ is a set of data.*

We assume that once an FB generates an action from an event output interface and its associated output data interfaces, the connected input interfaces simultaneously receive the action. For instance, an action $(DL, EO, \{(G, true)\})$ represents an execution where DL triggers an event from EO, then SW receives this event on EI so that the value of the associated data interface G is updated.

An IEC 61499 application is modelled as a Labelled Transition System (LTS) [Kel76] consisting of states and transitions labelled with actions.

Definition 3.2.2 (Behavioural model) *The behaviour of an IEC 61499 application is modelled as an LTS (S, s^0, A, T), where:*

- *S is a set of states, and $s^0 \in S$ is the initial state,*
- *A is a set of actions,*
- *$T \subseteq S \times A \times S$ is a set of transitions.*

An IEC 61499 behavioural model describes the global behaviour of an application consisting of every possible sequence of actions. In practice, during runtime, an application generates a sequence of actions called an execution trace (or trace for short). A trace represents the runtime behaviour of an application.

Definition 3.2.3 *(Trace)* A trace σ of size n is a sequence of actions $a_1, a_2, ..., a_n$.

The set of all possible traces in an LTS $m = (S, s^0, A, T)$ is denoted as $t(m) = \{\sigma_1, \sigma_2, ..., \sigma_n\}$. A trace can be extracted from an LTS or generated using monitoring techniques. Monitoring does not require the application to be transformed into an LTS. It is also useful to capture the nondeterministic behaviour of an application influenced by the presence of sensor SIFBs interacting with the environment. An example of a trace for the application shown in Figure 3.2 is $(DL, EO, \{(Q, \mathit{false})\}), (SW, EO0, \varnothing), (SR, EO, \{(Q, \mathit{true})\}), (DL, EO, \{(Q, \mathit{true})\}), (SW, EO1, \varnothing), (SR, EO, \{(Q, \mathit{false})\})$.

3.3 Translation techniques

An LTS is generated by first transforming the application into an LNT specification. This specification consists of a main process representing the application structure and other processes representing the FBs inside that application. This section explains how to translate FBs into these processes. In this thesis, we focus on the translation of BFBs. The LNT process of a SIFB can be created by analysing its input and output events and data. As for CFBs, they can be flattened into BFBs and SIFBs before the translation [DV08].

The translation from a BFB into an LNT process relies on the ECC of that BFB. The idea is to translate ECC states into LNT subprocesses. The code inside each subprocess is generated according to the outgoing transitions in the corresponding ECC state. The target state of the transition is translated as a call to the next subprocess representing that state. We use this concept to formulate translation patterns in Table 3.2. In these patterns, any text in italic x denotes an insertion of the value of x into the LNT code. Also, the parameters of the subprocess definition are hidden for brevity (i.e., π `( ) [ ]`). The translation patterns are as follows:

(1) A state with an unguarded transition is translated into a subprocess with an internal action followed by a call to the next subprocess.

(2) A state with a transition guarded by an input event interface is translated into a subprocess with an action composed of that interface and its associated input data interfaces followed by a call to the next subprocess.

(3) A state with a transition guarded by data is translated into a subprocess with a conditional statement parameterised by the boolean expression specified in the transition's guard. The next subprocess is called if the condition is satisfied; otherwise, the current subprocess is called.

Table 3.2: Translation patterns from ECC to LNT

Pattern	ECC state	LNT
(1) State with an unguarded transition	$(se, 1, se')$	**process** se`[ ]( ) is` `i ;` se'`[ ]( )` **end process**
(2) State with a transition guarded by an event	$(se, [\![ei]\!], se')$	**process** se`[ ]( ) is` ei`(?`$w(ei)$`) ;` se'`[ ]( )` **end process**
(3) State with a transition guarded by data	$(se, [\![[B_{DA}]]\!], se')$	**process** se`[ ]( ) is` **if** `(`B_{DA}`) then` se'`[ ]( )` **else** se`[ ]( ) end if` **end process**
(4) State with a transition guarded by an event and data	$(se, [\![ei[B_{DA}]]\!], se')$	**process** se`[ ]( ) is` ei`(?`$w(ei)$`) ;` **if** `(`B_{DA}`) then` se'`[ ]( )` **else** se`[ ]( ) end if` **end process**
(5) State with sequential transitions	$(se_0, 1, se_1)$, $(se_1, 1, se_2)$, $(se_2, ..., ...)$	**process** se_0`[ ]( ) is i ;` se_1`[ ]( )` **end process** **process** se_1`[ ]( ) is i ;` se_2`[ ]( )` **end process** **process** se_2`[ ]( ) is ...` **end process**

(6) State with branching transitions	$(se_0, [\![ei_1]\!], se_1)$,
	$(se_0, [\![ei_2]\!], se_2)$,
	$(se_0, ..., ...)$
		process se_0`[ ]( ) is`
		select
		ei_1`(?`$w(ei_1)$`) ;` se_1`[ ]( )`
		`[]` ei_2`(?`$w(ei_2)$`) ;` se_2`[ ]( )`
		`[] ... ; ...`
		end select
		end process

(7) State with a transition to another state with operations	$(se_0, 1, se_1)$, where $se_1 = (p_1, eo_1)$, $(p_2, eo_2), ...$	**process** se_0`[ ]( ) is`
		`i ;` p_1 `;` eo_1`(!`$w(eo_1)$`) ;`
		p_2 `;` eo_2`(!`$w(eo_2)$`) ;`
		`...` `;` se_1`[ ]( )`
		end process

(4) A state with a transition guarded by an event and data is translated according to pattern (2) followed by pattern (3).

(5) A state with a sequence of transitions is translated into a chain of subprocesses.

(6) A state with multiple outgoing transitions is translated into a subprocess with a select statement, where each block of code in the statement represents an outgoing transition.

(7) A state with a transition outgoing to another state with operations is translated into a subprocess with (i) a sequence of assignments in the programs followed by (ii) output events and (iii) a call to the next subprocess.

Example. Listing 3.1 shows LNT processes of the blinking application generated using the translation patterns. Process `DL` is created according to the behaviour of SIFB DL, which repeatedly triggers EO while updating the value of G. Process `SW`

Listing 3.1: SW and SR processes

```
1  process DL [EO: any] (G:bool) is
2    loop EO (?G) end loop
3  end process
4
5  process SW [EI, EO0, EO1: any] (G: bool) is
6    EI(?G) ;
7    if (G) then EO1 ; G1 [EI, EO0, EO1] (G)
8    else EO0 ; G0 [EI, EO0, EO1] (G) end if
9  end process
10 process G1 [EI, EO0, EO1: any] (G: bool) is
11   i ; SW [EI, EO0, EO1] (G)
12 end process
13 process G0 [EI, EO0, EO1: any] (G: bool) is
14   i ; SW [EI, EO0, EO1] (G)
15 end process
16
17 process SR [S, R, EO: any] (Q: bool) is
18   S ; Q := true ; EO(!Q) ; SET [S, R, EO] (Q)
19 end process
20 process SET [S, R, EO: any] (Q: bool) is
21   R ; Q := false ; EO(!Q) ; RESET [S, R, EO] (Q)
22 end process
23 process RESET [S, R, EO: any] (Q: bool) is
24   S ; Q := true ; EO(!Q) ; SET [S, R, EO] (Q)
25 end process
```

is translated from the initial state of BFB SW using patterns (4) and (7). This state has two outgoing transitions guarded by events and data. Therefore, as described in pattern (4), it is translated as a process with an action (line 6) followed by a conditional statement (lines 7 and 8). Pattern (7) is then applied to insert the target states' events into the statement (`EO1` in line 7 and `EO0` in line 8). Subprocesses `G1` and `G0` (lines 10 to 15) are generated according to pattern (1) because they correspond to ECC states with unguarded transitions. The initial state of SR is translated into process `SR` (lines 17 to 19) with patterns (2) and (7). The process begins with `S`, which is the event guard on the transition, followed by `Q := true` and `EO(!Q)`, which are obtained from the ECC state's operation. Afterwards, a call is made to the next process `SET`, which represents ECC state SET. The next two processes (lines 20 to 25) are generated using the same patterns as `SR`.

3.4 LTS generation

Once the application's FBs have been translated into LNT processes, the next step is to build the main process by creating a parallel composition of the generated processes. This parallel composition can be automatically created by analysing the application's structure.

Listing 3.2: Skeleton of an LNT main process

```
process main [Events : any] is
  var Variables : bool in
    par Parallel end par
  end var
end process
```

Algorithm 1: LNT main process generation

Inputs : $iec = (FB, EC, DC)$
Output : $Events, Data, Parallel$

$EO_{temp},\ DO_{temp},\ separator\ :=\ \varnothing,\ \varnothing,\ ""$
foreach $(eo, ei) \in EC$ **do** // iterate through the set of event connections
  if $eo \notin EO_{temp}$ **then** // check if the event has been added
    // concatenate the event connection to the string of events
    $Events\ :=\ Events + separator + getFB(eo, FB) + "\_" + eo$
    $separator\ :=\ ",\ "$
    $EO_{temp}\ :=\ EO_{temp} \cup eo$
$separator\ :=\ ""$
foreach $(do, di) \in DC$ **do** // iterate through the set of data connections
  if $do \notin DO_{temp}$ **then** // check if the data has been added
    // concatenate the data connection to the string of data
    $Data\ :=\ Data + separator + getFB(do, FB) + "\_" + do$
    $separator\ :=\ ",\ "$
    $DO_{temp}\ :=\ DO_{temp} \cup do$
$separator\ :=\ ""$
foreach $fb \in FB$ **do** // iterates through the set of FBs
  $FBsync\ :=\ getSync(fb, EC)$
  $FBevents\ :=\ fb.itf.EI \cup fb.itf.EO$
  $FBdata\ :=\ fb.itf.DI \cup fb.itf.DO$
  // concatenate the synchronisation set and process call to the end of the string for parallel composition
  $Parallel\ :=\ Parallel + separator\ +\ FBsync\ +\ "\ \rightarrow\ "$
  $+\ fb\ +\ "["+\ FBevents\ +\ "]"\ +\ "\langle" + FBdata + "\rangle"$
  $separator\ :=\ "\ ||\ "$

The objective is to complete the LNT code shown in Listing 3.2 using the components of the IEC 61499 application. *Events* consists of event declarations that are used in the main process. *Variables* is composed of variable declarations, and *Parallel* is the parallel composition of the processes that were previously generated using the translation techniques.

Algorithm 1 describes the generation of the LNT main process. It takes an IEC 61499 application as input. The algorithm returns *Events*, *Variables*, and *Parallel*

Listing 3.3: LNT specification of the IEC 61499 blinking application

```
1  process main [DL_EO, SW_EO0, SW_EO1, SR_EO: any] is
2    var SR_Q : bool in
3      par
4        DL_EO -> DL[DL_EO](SR_Q)
5      || DL_EO, SW_EO0, SW_EO1 -> SW[DL_EO, SW_EO0, SW_EO1](SR_Q)
6      || SW_EO0, SW_EO1 -> SR[SW_EO0, SW_EO1, SR_EO](SR_Q)
7      end par
8    end var
9  end process
```

as output. These outputs are strings that can be used to complete the LNT code shown in Figure 3.2. Function $getFB$ returns the FB of a given interface. Function $getSync$ returns the event interfaces of an FB that are connected to other FBs.

The algorithm first iterates through the set of event connections (lines 4 to 8). An output event interface is appended to $Events$ in each iteration (line 5). The same steps are required to obtain $Variables$ string by iterating through DO_{temp}. Four components are combined for each FB in order to construct the $Parallel$ string (lines 16 to 21). These are the synchronisation set, the name of the FB, the FB's events, and the FB's data (line 20). A process's synchronisation set is computed according to the event interfaces of the corresponding FB that are connected to other FBs. This computation is represented with function $getSync$, which takes as input an FB and the set of all event connections.

Example. Listing 3.3 presents the LNT main process of the blinking application. The process's parameters consist of events that can be triggered by the application (line 1). Note that events and data are appended with the corresponding FBs as prefixes to make them unique (e.g., `DL_EO` is event `EO` from FB `DL`). `Q`, which is the only data in the application, is declared as an LNT variable (line 2). A parallel composition is then built from the translated processes and synchronisation sets. Each of these processes represents an FB, and the synchronisation sets are obtained from the connections between FBs. For instance, `DL_EO` represents the connections between FBs DL and SW. Therefore, this connection is inserted in the synchronisation sets of both `DL` and `SW`.

The completed LNT specification is compiled into an LTS shown in Figure 3.3. The LTS shows all possible sequences of actions that the application can execute. For instance, after $(DL, EO, \{(Q, false)\}$ and $(SW, EO0, \varnothing)$ are executed, there are two possible actions to be executed next. These actions are either $(DL, EO, \{(Q, false)\}$ or $(SR, EO, \{(Q, false)\}$. The current LTS does not show the probability of executing

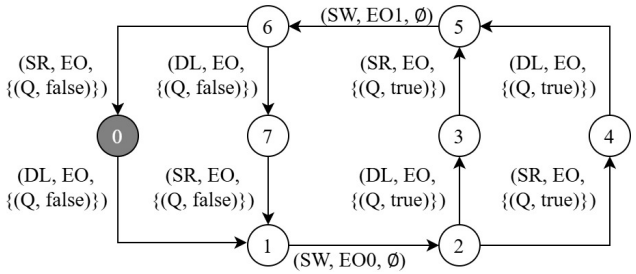


Figure 3.3: Blinking application LTS

these actions. The next chapter describes how to compute probabilistic values on the transitions using an LTS and an execution trace as input.

3.5 Concluding remarks

This chapter has explained the formal modelling of IEC 61499 applications. An IEC 61499 behavioural model is defined as a transition system labelled with the application's events and data (i.e., actions). The creation of this model involves translating each application's FB into an LNT process and subprocesses. A main process is then built and compiled into an LTS. The approach is applied to a blinking application to show that the model can describe all possible sequences of actions that the application can trigger.

4

Probabilistic model checking of IEC 61499 applications

This chapter presents the Probabilistic Model Checking (PMC) approach for IEC 61499 applications. The objective is to analyse the quantitative aspects of the applications by taking into account the likelihood of executions originating from the environment. The key idea is to enrich the behavioural models generated during design time (using formal modelling techniques) with execution traces monitored during runtime. This results in new models called Probabilistic Transition Systems (PTSs), which are LTSs extended with probabilistic values on their transitions. PTSs inform the probabilities of sequences of actions that the applications can trigger.

Figure 4.1 presents the approach overview. In steps (1) and (2), an LTS model is generated from the application. In step (3), a monitoring FB is synthesised and integrated into the application. Step (4) executes the application such that the monitor can start recording execution traces. The trace and the LTS are used in step (5) to compute a PTS model. In step (6), the CADP model checker [Gar+13] is used to check probabilistic properties, expressed as an MCL formula, on the PTS. CADP returns (i) a verdict and (ii) the probability of executing the sequences specified in the property. An industrial manufacturing application is used as a case study to illustrate our approach. The experimental results show that the approach helps in analysing and improving the system's productivity.

The structure of this chapter is as follows. Section 4.1 introduces the drilling station and its behavioural model. Section 4.2 explains the monitoring techniques. Section 4.3 describes how to compute probabilistic models. Section 4.4 presents the tool support. Section 4.5 shows the experimental results. Section 4.6 concludes.

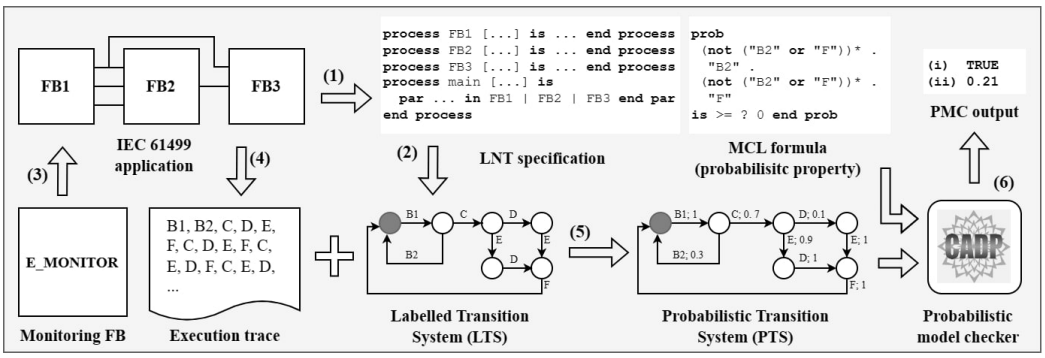
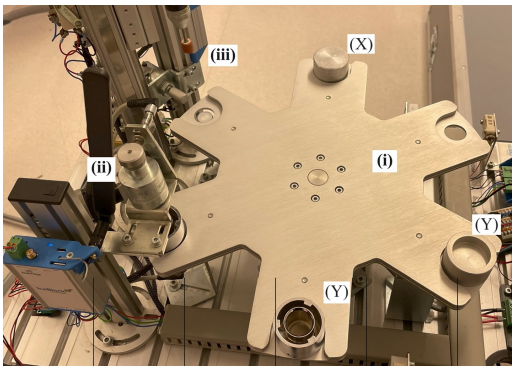
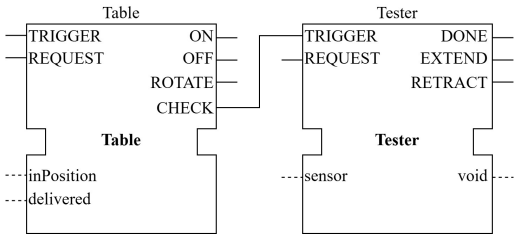


Figure 4.1: Probabilistic model checking for IEC 61499



(a) Mechatronic components



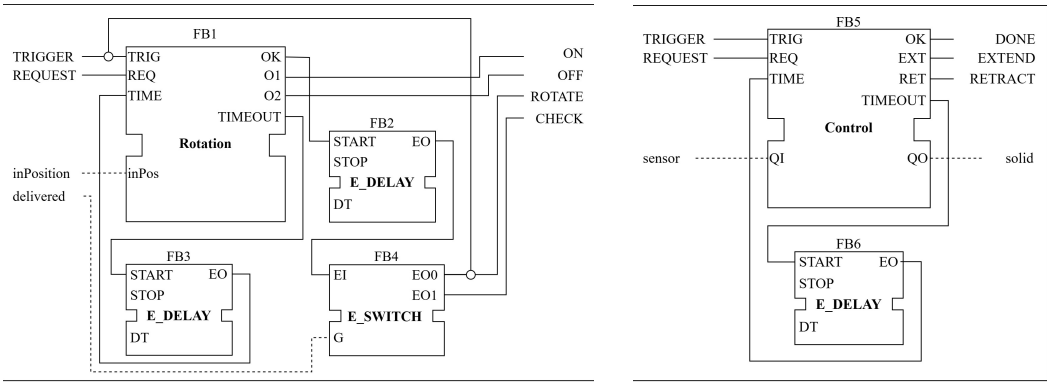
(b) Application

Figure 4.2: Drilling station

4.1 Case study

An IEC 61499 drilling station [XPV21] is used to illustrate the probabilistic model checking approach. As shown in Figure 4.2a, the system consists of three mechatronic components: (i) a table, (ii) a tester, and (iii) a drill. These components are considered smart, i.e., equipped with their own control devices that implement basic operations. The table component undergoes rotation from one fixed position to another. A complete cycle is achieved when the table rotates six times. Whenever a material is in the loading position, the table rotates to align it under the tester component. The tester then checks that the material has been drilled. If it has already been drilled, the drill component is not activated. Otherwise, the drill component is activated to drill the material as soon as its sensor detects the presence of the material beneath it. Materials that have not been drilled are called *solid* materials (X), while those that have been drilled are called *void* materials (Y).

There are operators who place solid materials on the table and take the void ones. These operators, who can either be humans or robots, are not part of the system.



(a) *Table* **(b)** *Tester*

Figure 4.3: *Table* and *Tester* CFBs

Instead, we consider their interaction with the drilling station as a nondeterministic aspect of the environment that can influence the system. For instance, if they do not place any material on the table, the drill will not be used even though the drilling station is running. Ideally, the system should drill as many materials as possible during its execution to optimise productivity.

The IEC 61499 application for the drilling station consists of 21 FBs. However, we focus on *Table* and *Test* CFBs depicted in Figure 4.2b. They are chosen because their input interfaces receive events and data from the sensor SIFBs associated with the environment. For instance, when the value of *delivered* in *Table* is *true*, it means that a material is detected in the loading position. It is not necessary to consider other FBs that have deterministic behaviour with respect to the environment. For example, when there is solid material under the drill component, it is certain that the drilling process, described by the respective FBs, will start. The networks of FBs inside *Table* and *Test* are shown in Figure 4.3. *ON* and *OFF* trigger events to start and stop the table rotation. *CHECK* is triggered when there is a material; otherwise, *ROTATE* is triggered. *EXTEND* and *RETRACT* are triggered sequentially to move the cylinder and check the material type. *DONE* is triggered with $void = false$ if it is a solid material; otherwise, $void = true$.

Modelling. The first step in modelling an IEC 61499 application is to translate the application into an LNT specification. We use the translation techniques explained in Chapter 3 to obtain the LNT processes shown in Listing 4.1. The subprocesses are not shown for brevity. Processes `Table` and `Tester` consist of parallel compositions between the FBs in *Table* and *Tester* CFBs, respectively. Another parallel composition is created in the main process to represent the drilling station application. Some LNT actions (e.g., `FB1_OK`) are hidden using **`hide`** construct (e.g., line 3) because they are not observable at the application level. LNT actions representing input

Listing 4.1: Drilling station LNT specification

```
process Table [FB1_TRIG, FB1_REQ, FB1_O1, FB1_O2,
FB4_EO0, FB4_EO1: any] (FB1_inPos, FB4_G: bool) is
  hide FB1_OK, FB1_O1, FB1_O2, FB1_TIMEOUT,
  FB2_STOP, FB2_EO, FB3_STOP, FB3_EO, FB4_EO0, FB4_EO1: any in
    par
      FB4_EO0, FB3_EO, FB1_OK, FB1_O1, FB1_O2, FB1_TIMEOUT ->
      FB1 [FB1_TRIG, FB4_EO0, FB1_REQ, FB3_EO,
      FB1_OK, FB1_O1, FB1_O2, FB1_TIMEOUT] (FB1_inPos)
    || FB1_OK, FB2_EO -> FB2 [FB1_OK, FB2_EO]
    || FB1_TIMEOUT, FB3_EO -> FB3 [FB1_TIMEOUT, FB3_EO]
    || FB2_EO, FB4_EO0, FB4_EO1 -> FB4 [FB2_EO, FB4_EO0, FB4_EO1] (FB4_G)
    end par
  end hide
end process

process Tester [FB5_TRIG, FB5_REQ, FB5_OK, FB5_EXT,
FB5_RET: any] (FB5_QI, FB5_QO: bool) is
  hide FB6_EO, FB5_TIMEOUT: any in
    par
      FB6_EO, FB5_TIMEOUT -> FB5 [FB5_TRIG, FB5_REQ, FB6_EO,
      FB5_OK, FB5_EXT, FB5_RET, FB5_TIMEOUT] (FB5_QI, FB5_QO)
    || FB5_TIMEOUT, FB6_EO -> FB6 [FB5_TIMEOUT, FB6_EO]
    end par
  end hide
end process

process main [Table_ON, Table_OFF, Table_ROTATE, Table_CHECK,
Tester_DONE, Tester_EXTEND, Tester_RETRACT: any] is
  hide Table_TRIGGER, Table_REQUEST, Tester_REQUEST: any in
    var Table_inPosition, Table_delivered, Tester_sensor, Tester_void: bool in
      par Table_CHECK in
        Table [Table_TRIGGER, Table_REQUEST, Table_ON, Table_OFF,
        Table_ROTATE, Table_CHECK] (Table_inPosition, Table_delivered)
      || Tester [Table_CHECK, Tester_REQUEST, Tester_DONE, Tester_EXTEND,
         Tester_RETRACT] (Table_inPosition, Table_delivered)
      end par
    end var
  end hide
end process
```

events (e.g., TABLE_TRIGGER) are also hidden because we only model sequences of events and data triggered (not received) by the application FBs.

The LNT specification is then compiled into an LTS presented in Figure 4.4. This LTS shows sequences of actions that the application can trigger. Every action corresponds to a mechatronic component's physical activity. For instance, let us consider the following sequence of activities: (i) the table starts moving, (ii) the table stops moving, (iii) a material is detected, (iv) the tester's cylinder extends, (v) the tester's cylinder retracts, (iv) a solid material is detected because the cylinder collides with the top surface of the material. This sequence corresponds to the following sequence of actions: $(Table, ON, \varnothing), (Table, OFF, \varnothing), (Table, CHECK, \varnothing), (Tester, EXTEND, \varnothing), (Tester, RETRACT, \varnothing), (Tester, DONE, \{(void, false)\})$.

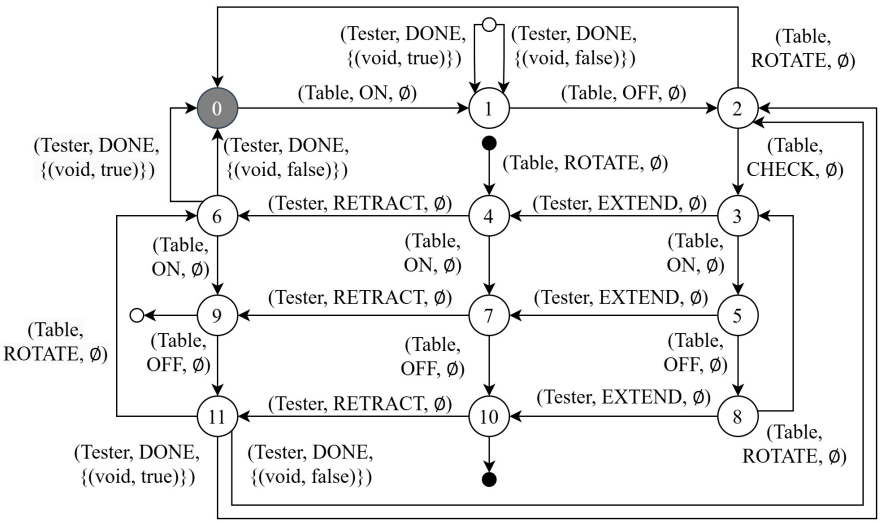


Figure 4.4: LTS of the drilling station

4.2 Monitoring techniques

The goal of monitoring the application is to obtain execution traces for computing probabilistic models. To do so, a monitoring FB is synthesised and integrated into the application. This FB is a BFB with input interfaces that receive actions triggered by the application and an ECC that records these actions.

The synthesis and integration of a monitoring FB is described in Algorithm 2. The algorithm iterates through the output interfaces of the application's FBs to create the monitoring FB components. For every output event interface, the following components are created: input event interface (line 8), ECC transition guard (line 9), ST program (line 11), ECC state (line 13), ECC transitions (line 16), event connections (line 17). For every output data interface, a monitoring input data interface is created (line 20), and a new connection is made (line 21).

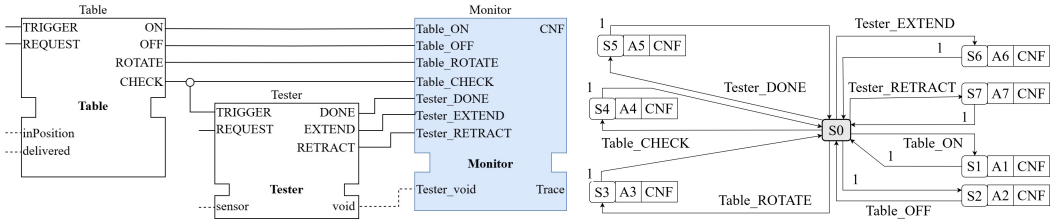
Example. Figure 4.5 shows the monitored drilling station application and the ECC inside the monitoring FB. Eight input interfaces are created and connected to the output interfaces of the application. The ECC describes how an action is recorded every time an event is received. For instance when $Table\_ON$ receives an event, the current state moves from $S0$ to $S1$. In this state, $A1$ is executed to record $Table\_ON$, which encodes as an action $(Table, ON, \varnothing)$. Then, an event is triggered from CNF to update the value of $Trace$, and the current state moves back to the initial state. The ST program in $A5$ is different from others because $DONE$ is associated with a data interface $void$. As seen in lines 1 to 5, the value of the data in the action is assigned according to the value of the interface.

Algorithm 2: Synthesis and integration of a monitoring FB

Inputs : $iec = (FB, EC, DC)$

Output : $fb_m = (itf_m, DV_m, P_m, ecc_m)$

1 $itf_m := (EI, \{CNF\}, DI, \{(Trace, \varnothing)\}, AI, \{(CNF, Trace)\})$
2 $DV_m := \{(Index, 0)\}$ // create an index variable
3 $ecc_m := (Se, se^0, Ge, Te)$
4 **foreach** $fb = (itf, DV, P, ecc) \in FB$ **do** // iterate through the set of FBs
5 $itf := (EI', EO', DI', DO', AI', AO')$
6 **foreach** $eo' \in EO'$ **do** // iterate through output event interfaces
7 $ei := fb + \text{“\_”} + eo'$
8 $EI := EI \cup \{ei\}$ // create an input event interface
9 $Ge := Ge \cup \{[\![ei]\!]\}$ // insert an ECC transition guard
10 $p := createST(fb, ei)$ // create an ST program to record the input
11 $P_m := P_m \cup \{p\}$
12 $se := (p, CNF)$
13 $Se := Se \cup \{se\}$ // create an ECC state operation
14 $te_1 := (se^0, [\![ei]\!], se)$
15 $te_2 := (se, 1, se^0)$
16 $Te := Te \cup \{te_1, te_2\}$ // create ECC transitions
17 $EC := EC \cup \{(eo', ei)\}$ // connect the event interfaces
18 **foreach** $do' \in DO'$ **do** // iterate through output data interfaces
19 $di := fb + \text{“\_”} + do'$
20 $DI := DI \cup \{di\}$ // create an input data interface
21 $DC := \{(do', di)\}$ // connect the data interfaces



(a) Application **(b)** ECC

A1
```
1 | Trace[idx] := "Table_ON";
2 | Index ++
```

A2
```
1 | Trace[Idx] := "Table_OFF";
2 | Index ++
```

A3
```
1 | Trace[Idx] := "Table_ROTATE";
2 | Index ++
```

A4
```
1 | Trace[Idx] := "Table_CHECK";
2 | Index ++
```

A5
```
1 | if (void) {
2 |  Trace[idx]:="Tester_DONE_void_true";
3 | } else {
4 |  Trace[idx]:="Tester_DONE_void_false";
5 | }
6 | Index ++
```

A6
```
1 | Trace[Idx] := "Tester_EXTEND";
2 | Index ++
```

A7
```
1 | Trace[Idx] := "Tester_RETRACT";
2 | Index ++
```

(c) ST programs

Figure 4.5: Drilling station application extended with a monitoring FB

4.3 Probabilistic model computation

An LTS is a model of the application that represents all possible sequences of actions. This model can be extended into a probabilistic model such that it also represents the likelihood of those sequences to be executed by the application. In this thesis, we adopt the Probabilistic Transition System (PTS) [LS91] as a probabilistic model of an IEC 61499 application. A PTS is composed of the same elements as an LTS with the addition of a probability in each of its transitions. The sum of probabilities of transitions outgoing from a given state is equal to 1, and by default, they have the same value (i.e., *equiprobable*).

Definition 4.3.1 (Probabilistic transition system) *A PTS is an LTS* (S, s^0, A, T_p) *in which transitions are extended with probabilities such that* $T_p \subseteq S \times A \times p \times S$*, where each* $p \in [0, 1]$ *is the probability of executing a transition from its source state.*

Algorithm 3: Computation of PTS

Inputs : $m = (S, s^0, A, T), \sigma \in t(m)$
Output : $m_p = (S_p, s_p^0, A, T_p)$

1 $s_p^0 := (s^0, 1)$ // set initial state counter to 1
2 $S_p := \{(s, 0) \mid s \in S \setminus \{s^0\}\} \cup \{s_p^0\}$ // set remaining state counters to 0
3 $T_p := \{(s, a, s', 0, 0.) \mid (s, a, s') \in T\}$ // set counters and probabilities to 0
4 $s_p := s_p^0$ // current state
5 **foreach** $a \in \sigma$, *in order* **do** // iterate through the actions in the trace
6 **let** $T'_p \subseteq T_p$ be the set of transitions outgoing from $(s, x) = s_p$ **in**
7 **foreach** $((s, x), a', (s', x'), y, p) \in T'_p$ **do**
8 **if** $a = a'$ **then** // transition's action = the current action in the trace
9 $y := y + 1$ // increment transition counter
10 $s_p := (s', x' + 1)$ // set the next state
11 $p := y \div x$ // compute the probability

Algorithm 3 takes an LTS and a trace as input to produce a PTS. The first four lines show the preliminary initialisation of variables. In a PTS, each state is extended with a counter x, whereas each transition has a counter y and a probability p. Counters represent how many times states/transitions have been traversed. They are initialised to 0 except for the initial state, 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 is completed. More precisely, the PTS is traversed by iterating

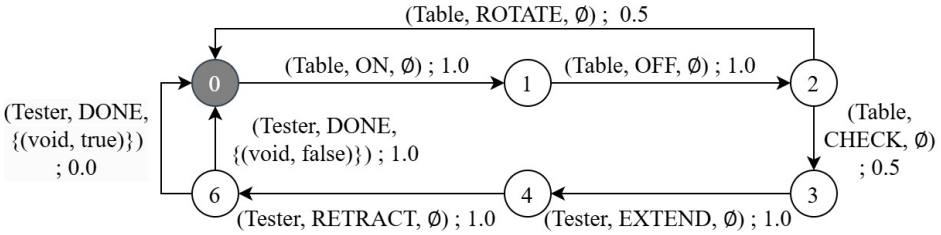


Figure 4.6: Example of drilling station PTS

over the trace starting from the first action (line 5). If an outgoing transition's label a' is the same as the current action a, the counter on that transition is updated, and the next state is selected (lines 8 to 10). Afterwards, the outgoing transition probability is updated by dividing its counter with the state's counter.

Example. Figure 4.6 shows an excerpt of PTS computed from the drilling station LTS in Figure 4.4 and a trace obtained using monitoring setup in Figure 4.5. In this case, the drilling station rotated six times (i.e., one cycle), and three solid materials were detected. As one of the observable results, the probabilities of transitions from states 2 to 0 with action $(Table, ROTATE, \varnothing)$ and 2 to 3 with action $(Table, CHECK, \varnothing)$ are the same. This example shows that observing the PTS allows us to see the probability of event sequences. However, techniques such as PMC are required for more specific results that are not straightforward to obtain manually. For instance, it is not straightforward to compute the probability of detecting three void materials in six rotations (i.e., a complete cycle) only by looking at the PTS. Moreover, a complex model may contain millions of states. Hence, it would be impossible to analyse the PTS visually.

4.4 Tool support

We have developed a tool to compute the PTS of a given LTS and an execution trace [1]. This tool is the implementation of Algorithm 3 and is written in Java. It takes as input an LTS and a trace in textual format as shown in Figure 4.7. The generated PTS, along with a probabilistic property written in MCL5, can be used as input for PMC using the CADP toolbox [Gar+13].

[1] available online at `https://gitlab.inria.fr/ifaqriza/pts-computation`

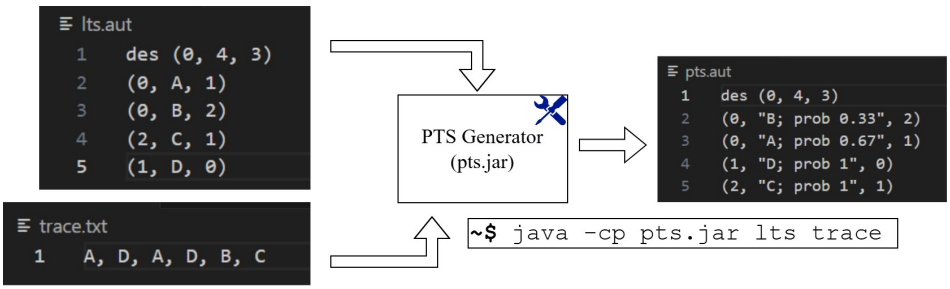


Figure 4.7: Tool support for computing PTSs

4.5 Experimental results

This experiment [2] aims to show that PMC is helpful in analysing the impact of the environment for optimising the system's productivity. For this, we define a scenario where the environment is represented by different strategies for placing and taking materials on the drilling station table. The system is executed according to these strategies to obtain several execution traces for generating multiple PTS models. Several probabilistic properties are then checked on the models using the CADP model checker. CADP can return the probability of action sequences specified in each property (see Figure 4.1). The resulting probabilities are presented as charts that can be analysed to see which strategy is better for productivity and also to improve existing strategies.

Scenario. Two workers are involved as the drilling station operators. *Placer* puts solid materials on the table, while *Taker* picks the void ones. They have other responsibilities because the drilling station is part of a larger industrial system. Therefore, they are not always stationed at their location. The notion of environment varies depending on the system. In more complex systems, the role of Placer and Taker can be replaced by other external aspects (e.g., room temperature) that impact the system in a more complicated manner.

Two strategies in Figure 4.8 are chosen for this experiment. In S1, the void materials are taken individually, whereas in S2, they are all taken at once but after a longer period of time. The strategies are described as follows:

(S1) Placer places a solid material every 20 to 25 seconds, and Taker picks a void material every 35 to 40 seconds.
(S2) Placer places a solid material every 20 to 25 seconds, and Taker picks all void materials every 85 to 90 seconds.

[2]Available online at `https://gitlab.inria.fr/ifaqriza/pmc-experiments`

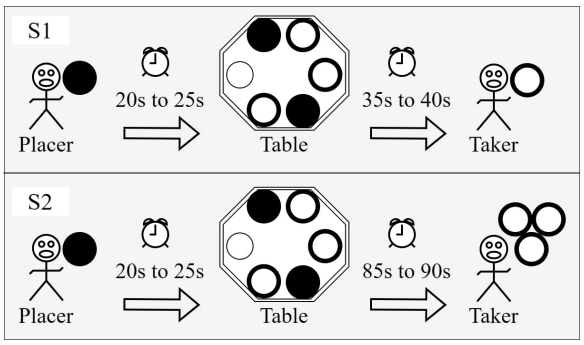


Figure 4.8: Strategies for placing and taking the materials

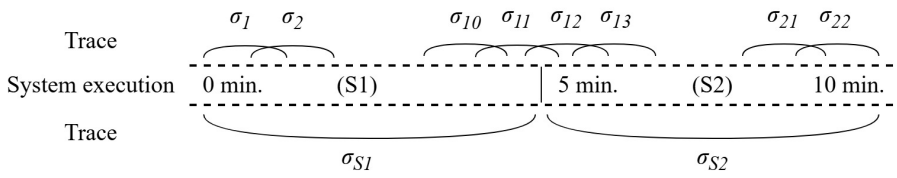


Figure 4.9: Drilling station experimental system execution

System execution. The system was executed and monitored for 10 minutes using the aforementioned strategies. Two main execution traces containing nearly 700 actions were obtained. More precisely, the system execution for this experiment is illustrated in Figure 4.9. S1 was applied in the first five minutes to get the first trace σ_{S1}, and for the second half, we used S2 to obtain σ_{S2}. These two traces are used to generate PTS models that can be checked as a summary of the system execution. In addition, the entire trace is cut into 22 pieces using a *sliding window* [MBB17] to generate several traces $\sigma_1, \sigma_2, ..., \sigma_{22}$. These traces are used to generate PTS models for checking the system execution over time as the Placer and Taker apply their strategies.

4.5.1 Probabilistic properties

We focus on the following probabilistic properties to measure the productivity of the system:

(P1) the probability of detecting a type of material,
(P2) the probability of having a certain number of materials in the system, and
(P3) the probability of detecting a solid material in several rotations.

P1 helps to analyse system productivity by showing how often a type of material is detected during system execution. Three MCL formulas associated with this property are presented in Listing 4.2. The first one is to check the probability of

Listing 4.2: MCL formulas for P1

```
1  prob (* empty place *)
2    (not ("Table_CHECK" or "Table_ROTATE"))* . "Table_ROTATE"
3  is >= ? 0 end prob
4
5  prob (* solid material *)
6    (not ("Table_CHECK" or "Table_ROTATE"))* . "Table_CHECK" .
7    (not ("Tester_DONE_void !TRUE" or "Tester_DONE_void !FALSE"))* .
8    "Tester_DONE_void !FALSE"
9  is >= ? 0 end prob
10
11 prob (* void material *)
12   (not ("Table_CHECK" or "Table_ROTATE"))* . "Table_CHECK" .
13   (not ("Tester_DONE_void !TRUE" or "Tester_DONE_void !FALSE"))* .
14   "Tester_DONE_void !TRUE"
15 is >= ? 0 end prob
```

detecting an empty place, whereas the second and third ones are for solid and void materials. The regular expression inside each formula specifies the sequences of actions corresponding to the detection of a material. For instance, the second formula specifies a sequence where $(Table, CHECK, \varnothing)$ (i.e., there is material) is followed by $(Tester, DONE, \{(void, false)\})$ (i.e., the material is solid).

Listing 4.3: Excerpt of MCL formulas for P2

```
1  prob
2    loop (x, y: Nat := 0) : (z: Nat) in
3      if (x = 6) then exit (y)
4      else
5        ("Table_ROTATE" or "Tester_DONE_void !TRUE").continue(x + 1, y)
6      | ("Tester_DONE_void !FALSE").continue(x + 1, y + 1)
7      | not ("Table_ROTATE" or "Tester_DONE_void !FALSE"
8        or "Tester_DONE_void !FALSE").continue(x , y)
9      end if
10   end loop .
11   if z <> 3 then false end if
12 is >= ? 0 end prob
```

In P2, the number of materials in the system means the number of materials after a full cycle (i.e., six rotations). The results can be used as a summary for evaluation purposes. There are 21 MCL formulas for this property to check the probabilities of having zero to six materials for all three types. For example, Listing 4.3 shows a formula to check the probability of having three solid materials in the system [3]. A combination of loop and conditional operators is used to represent sequences

[3] Lines 5 to 8 can be replaced with sequences that represent the detections of other types of material, and the number 3 at line 11 can be replaced with 0 to 6 to check the probability of having a different number of materials.

of actions. The loop iterates through transitions in the PTS until sequences that represent a full cycle are obtained (line 3). The condition ensures that only sequences with three detections of solid material are returned (line 11).

Listing 4.4: Excerpt of MCL formulas for P3

```
1  prob
2    ( (not ("Table_CHECK" or "Table_ROTATE"))* . (("Table_CHECK" .
3      (not ("Tester_DONE_void !TRUE" or"Tester_DONE_void !FALSE"))* .
4     "Tester_DONE_void !TRUE") | ("Table_ROTATE"))
5    ) {0 ... 10} .
6    (not ("Table_CHECK" or "Table_ROTATE"))* . "Table_CHECK" .
7    (not ("Tester_DONE_void !TRUE" or "Tester_DONE_void !FALSE"))* .
8    "Tester_DONE_void !FALSE"
9  is >= ? 0 end prob
```

P3 returns the probability of detecting a solid material after some rotations. It also shows how many rotations are required to make the probability of detecting a solid material close to 100% (e.g., more than 90%). There are 21 MCL formulas for this property to check from after 0 to 20 rotations. The formula to check the probability of detecting a solid material after ten rotations is shown in Listing 4.4 [4]. Here, the first part refers to 10 detections of an empty place or a void material in a row (lines 2 to 5). The second part signifies the detection of a solid material (lines 6 to 8).

MCL formulas for P1 are used to check PTS models generated using traces $\sigma_1, \sigma_2, ..., \sigma_{22}$. This allows us to see the probabilities of detecting types of material over time. On the other hand, formulas for P2 and P3 are checked on models computed from σ_{S1} and σ_{S2} to show a summary of how the two strategies can impact the probabilities of material detection.

4.5.2 Results and discussion

The probabilistic properties are checked on the PTS models computed from the execution traces. Since there are several formulas and models, the results are presented using charts to analyse the impact of strategies in terms of productivity.

Figure 4.10 shows the probability of detecting materials (P1). Window (x-axis) represents the PTS models generated over time using $\sigma_1, \sigma_2, ..., \sigma_{22}$. The probability of detecting void materials increases up to 77% when S1 is applied. It decreases and remains close to 50% in the second half when the strategy is changed to S2. In

[4] The number 10 at line 5 can be replaced with 0 to 20 to check the probability of detecting a solid material after a different number of rotations.

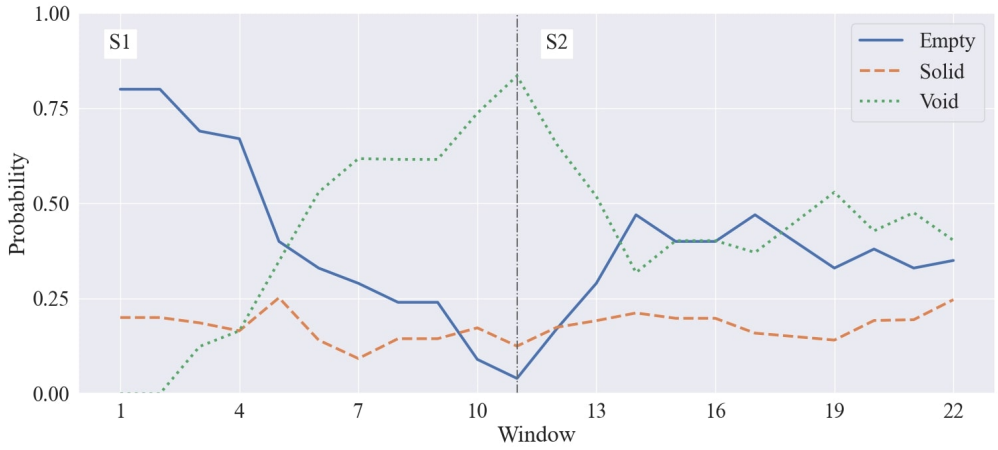


Figure 4.10: Probability of detecting different types of material over time

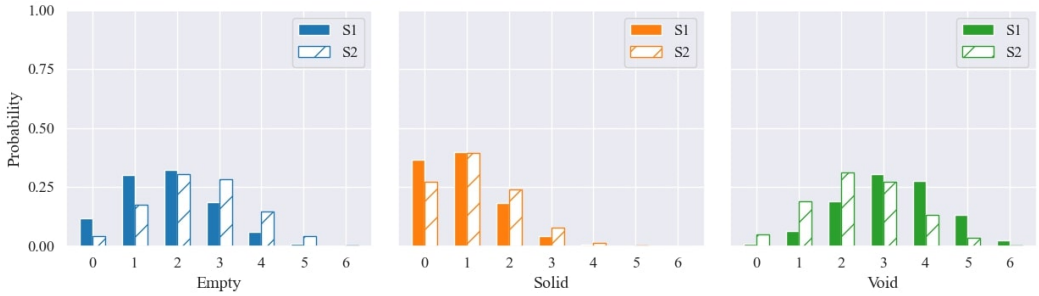


Figure 4.11: Probability of the numbers of materials in the system

contrast, the probability of empty places gets lower when using S1 until reaching 5%; then, it returns to about 45% when S2 is used. The probability of solid material detection stays just under 25% in both strategies.

Figure 4.11 shows the probability of having certain numbers of materials in the system (P2). The probability of empty places is overall higher using S2 compared to S1; with S2, there is 5% probability that there are five empty places. The probability of having solid materials on the table is also higher when S2 is used, with the probability of having two of them close to 25%. The probability of void materials in the system is higher using S1. Furthermore, there is a 4% probability that void materials fully occupy the table.

Figure 4.12 shows the probability of detecting a solid material in several rotations (P3). S2 requires fewer rotations to get a higher probability. More precisely, only 12 rotations are needed to get the probability above 90%, while S1 requires 16 rotations. As highlighted, the peak difference of probabilities between the two strategies occurs in four rotations (57% and 66%, respectively, for S1 and S2).

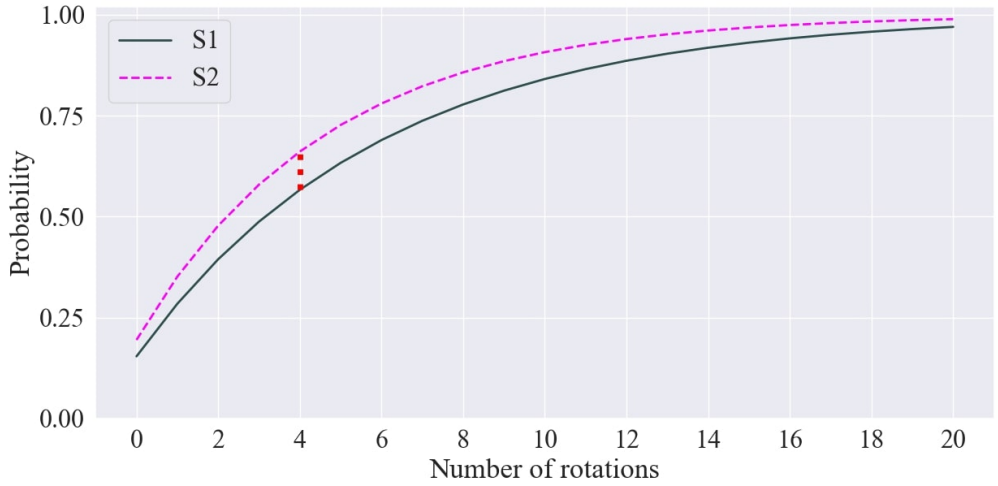


Figure 4.12: Probability of detecting a solid material in several rotations

The results show that the two strategies representing two environments have different impacts on the drilling station. There are two important points from this experiment. Firstly, S2 is better than S1 for optimising productivity because the probability of detecting void materials is lower, allowing solid materials to be placed on the table. Secondly, the probability of detecting empty places is also higher in S2. Therefore, Placer should put solid materials on the table more frequently. For instance, we may propose a new strategy where solid material is placed every 10 to 15 seconds. Overall, analysing the results of PMC can help to decide which environment (i.e., strategy) is better for optimising the quantitative aspects of the system (i.e., productivity).

4.6 Concluding remarks

This chapter provides an answer to **RQ1** stated in Chapter 1. PMC can be used to formally verify and analyse the quantitative aspects of IEC 61499 applications originating from the environment. The approach begins with building the probabilistic model of the application. This model, called PTS, contains the runtime information of the system because it has been enriched with an execution trace gathered using monitoring techniques. It allows verifying the probability of action sequences that the application can trigger.

In our experiment, PMC is applied to analyse an IEC 61499 drilling station. The environment is represented as operators' strategies when interacting with the system, and the objective is to optimise productivity. Several probabilistic properties were

checked on the system's generated probabilistic models. We show that the results visualised using charts are helpful for exhaustive analysis of the impact of strategies on the system. This allows us to choose which strategy is better and propose new strategies to increase productivity.

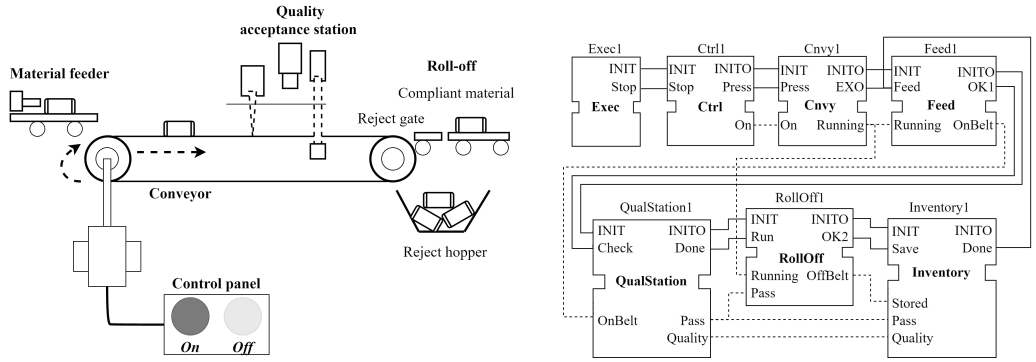
5

Runtime enforcement for IEC 61499 applications

This chapter presents the runtime enforcement approach for IEC 61499 applications. The goal is to evolve the initial application according to some given requirements. The idea is to insert enforcers in the form of BFBs into the application to modify its runtime behaviour as specified in the requirements. Our approach avoids manual modifications that require considerable effort and may introduce errors.

Our approach begins with specifying the requirements using a specification language that we proposed called contract automata. A contract automaton is a state machine in which its transitions can be typed to apply specific modifications to the events and data triggered by the application. This automaton is then used to synthesise an enforcer as a BFB. Finally, the new FB is integrated into the application to execute according to the requirements. The approach is demonstrated using an IEC 61499 conveyor test station. We show that our method conforms to *soundness* and *transparency*, and *deadlock-freedom* can also be verified using model checking. A tool for automating the synthesis of enforcers has been developed. In addition, experiments were performed to evaluate the performance of enforcers by measuring the execution time of several applications before and after the integration of enforcers.

This chapter is organised as follows. Section 5.1 introduces the case study. Section 5.2 explains the IEC 61499 runtime enforcement problem. Section 5.3 defines the contract automata language. Section 5.4 describes the enforcer synthesis and integration. Section 5.5 explains the preserved runtime enforcement characteristics. Section 5.6 presents the tool support. Section 5.7 shows the experimental results. Section 5.8 concludes.



(a) Design **(b)** Application

Figure 5.1: Conveyor test station

5.1 Case study

This section introduces an IEC 61499 conveyor test station [ZL14] as a case study to be used for illustrating the runtime enforcement approach. Minor modifications were made to facilitate the explanation of our approach.

Figure 5.1a presents the physical design of the running example. The system aims to check the qualities of industrial materials passing through a conveyor belt. Firstly, a conveyor is connected to a control panel where the user can either start or stop the conveyor (i.e., by pressing on or off buttons). Then, the component feeder is in charge of transferring materials onto the conveyor. Next, a quality acceptance station evaluates the materials as they pass through. Lastly, depending on the test results, the roll-off mechanism allows the materials to be distributed onto the next industrial process or dropped into a hopper by opening the reject gate.

The IEC 61499 application of the conveyor test station is presented in Figure 5.1b. $Cnvy1$ and $Ctrl1$ correspond to the conveyor and its control panel. $Feed1$ represents the material feeder, $QualStation1$ deals with the quality acceptance station, and $RollOff1$ handles the roll-off mechanism. $Exec1$ and $Inventory1$ correspond to the initialisation of the application and the database which stores the materials data. In all these FBs, event and data interfaces are associated with the application's functionalities. For instance, the $Running$ data interface in $Cnvy1$ has a boolean value representing the conveyor belt's state. When the conveyor is running, then $Running = true$, otherwise $Running = false$. The $Quality$ data interface has an integer value representing the quality of the material. It ranges from 0 to 100, and the material is accepted if this value exceeds 50.

The application was designed by its authors following the IEC 61499 hierarchical design patterns [ZP13]. As a result, the application consists of seven CFBs with at least 35 FBs inside of them in total. A more comprehensive description of the conveyor test station can be found in [ZL14].

5.2 IEC 61499 runtime enforcement problem

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

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

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

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

5.3 Contract automata

We propose a specification language called contract automata to express the requirements of IEC 61499 applications. When there is a situation in which the application needs to adapt to the requirements, users can specify these requirements

Figure 5.2: Transition types in contract automata

as a contract automaton. The term contract means that the application will behave according to the automaton, whereas automata is a common terminology in runtime enforcement when defining specification languages. For instance, two notable languages are *Security Automata* [Sch00] and *Edit Automata* [LBW05]. Contract automata is an extended version of these two languages where transitions are typed to add expressiveness. The notions of *accepting* and *non-accepting* states are excluded because there is no need to specify the type of each state. Contract automata can be specified to buffer the application executions (i.e., actions). Users may also specify to discard or replace specific executions when necessary. Overall, in this work, contract automata is proposed as an expressive and intuitive runtime enforcement specification language for IEC 61499 applications.

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

5.3.1 Syntax

A contract automaton consists of states and transitions. The transitions are typed to specify unique modifications to the set of actions that the application can execute. In other words, when an action is executed, the type of transition corresponding to that action determines the modification to be applied. In Figure 5.2, these types are represented with different styles, and their descriptions are as follows:

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

Besides a type, a transition also has a guard. The set of all guards present in the automaton is denoted as G. A guard $g \in G$ is written $[\![(fb)e[B_D]]\!]$, where fb is an FB, e is an event, and B_D is a boolean expression over a set of data. The set of actions specified by G is denoted by function $Act : 2^G \to 2^\Sigma$. Every guard is a function $g : \Sigma \to \mathbb{B} \cup \{i\}$. For an action $a = (fb', e', D)$ and a guard $g = [\![(fb)e[B_D]]\!]$,

$g(a) = true$ iff $(fb' = fb) \wedge (e' = e) \wedge (D \models B_D)$, otherwise $g(a) = false$. The symbol i is returned when the input action $a \notin Act(G)$.

To illustrate guards, let us consider actions $a = (Ctrl1, Press, \{(On, true)\})$, $a' = (Feed1, OK1, \{(OnBelt, true)\})$, and a guard $g = [\![(Ctrl1)Press[On = false]]\!]$. $Act(G) = \{(Ctrl1, Press, \{(On, true)\}), (Ctrl1, Press, \{(On, false)\})$ is the set of specified actions. Here, $g(a) = false$ since $\{(On, true)\} \not\models (On = false)$, whereas $g(a') = i$ because $a' \notin Act(G)$.

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

Definition 5.3.1 *(Contract automaton)* A contract automaton is a tuple (S, s^0, Γ, G, T), where:

- S is a (finite) set of states, and $s^0 \in S$ is the initial state,
- $\Gamma = \{forward, discard, replace, buffer\}$ is the set of transition types,
- G is a set of guards and $Act : 2^G \rightarrow 2^\Sigma$ returns the set of specified actions,
- $T \subseteq S \times \Gamma \times G \times Act(G) \times S$ is a set of transitions and each transition $t = (s, \gamma, g, \alpha, s') \in T$, where:
  - $s, s' \in S$ are source and target states,
  - $\gamma \in \Gamma$ is the transition type,
  - $g \in G$ is the guard,
  - $\alpha \in Act(G)$ is a replacement action.

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

Example. Two examples called Automaton A and Automaton B are presented in Figures 5.3 and 5.4 to illustrate contract automata. Given the running example presented in Section 5.1, suppose the following functional requirements are desired:

(i) Pressing the *on* button when the conveyor is running or *off* when idle should not impact the application to prevent unexpected behaviour.
(ii) If the button *off* is pressed when there is still material on the conveyor, then the conveyor will only stop running when that material has passed through the roll-off mechanism to ensure that the conveyor is unoccupied when it stops.

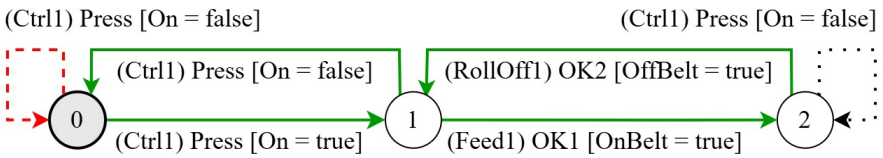


Figure 5.3: Automaton A with *discard* and *buffer* transitions

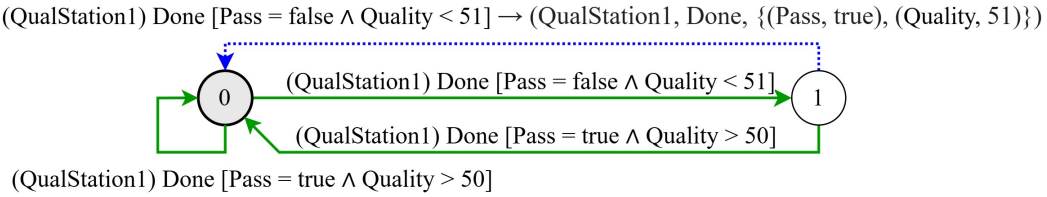


Figure 5.4: Automaton B with a *replace* transition

(iii) To avoid too many rejected materials, there should not be two rejections in a row. The second rejection is considered an acceptance instead, and the material's quality is set to 51.

Automaton A aims to satisfy (i) and (ii). For specifying requirement (i), from the initial state, we apply a correction by discarding the action satisfying the guard $[\![(Ctrl1)Press[On = false]]\!]$ to suppress the impact of a user pressing the off button when the conveyor is idle. Transitions typed as *discard* can be implicit when the source and target states are the same (i.e., self-loop). For instance, in state 1, it is not mandatory to specify a *discard* transition guarded with $[\![(Ctrl1)Press[On = true]]\!]$. This discard mechanism is explained further in the semantics part of this section. To satisfy (ii), we specify the transition in state 2 to buffer an action satisfying the guard $[\![(Ctrl1)Press[On = false]]\!]$. This is because in this state, action $(Feed, OK1, \{(OnBelt, true)\})$ has been executed (i.e., the material has been fed onto the conveyor) and action $(RollOff1, OK2, \{(OffBelt, true)\})$ has not been executed yet (i.e., the material has not passed through the roll-off mechanism).

Automaton B specifies requirement (iii). In state 1, when the action satisfying the guard $[\![(QualStation1)Done[Pass = false \wedge Quality < 51]]\!]$ (i.e., the quality station rejects the material) is executed for the second time, we use a replace transition to trigger a different action that is $(QualStation1, Done, \{(Pass, true), (Quality, 51)\})$ (i.e., the material is accepted and its quality is set to 51).

5.3.2 Semantics

The notion of buffer is needed to define the semantics of contract automata. The set of bags over the set of actions Σ is denoted by $\mathbb{G}_\Sigma$. A buffer $buf \in \mathbb{G}_\Sigma$ is a bag to store buffered actions. The following functions are used to represent the operations on this buffer. Functions $add : \mathbb{G}_\Sigma \times \Sigma \to \mathbb{G}_\Sigma$ and $remove : \mathbb{G}_\Sigma \times \Sigma \to \mathbb{G}_\Sigma$ are respectively used for adding and removing actions from the buffer. Function $Find : S \times \mathbb{G}_\Sigma \to \Sigma^*$ finds a sequence of actions in the buffer that subsequently satisfies the guards of $forward$ transitions from a given starting state. Function $Reach : S \times \Sigma^* \to S$ returns a reachable state from a given starting state and a sequence of actions.

A transition system is used to describe the semantics of contract automata. It relies on the notion of configuration. The set of configurations in a contract automaton is defined as $Conf = S \times \mathbb{G}_\Sigma$, where S is the set of states and $\mathbb{G}_\Sigma$ is the set of buffers. A transition $(s, buf) \xrightarrow{a/o} (s', buf')$ indicates a move from configuration (s, buf) to configuration (s', buf') while taking an input $a \in \Sigma$ and returning an output $o \in \Sigma^*$.

Table 5.1 presents the transition rules between configurations in contract automata. Each rule defines the behaviour of the application every time an action is executed. We use examples of contract automata in Figures 5.3 and 5.4 (i.e., Automaton A and Automaton B) to illustrate these rules in the following explanations:

- f_0 states that if every transition guard returns i when evaluating an action, then, this action is not modified, and the configuration does not change. This is the case when an action $(Exec1, INIT, \varnothing)$ is executed from any state in Automaton A, this action is forwarded, and the current state does not move.
- f_1 states that when an action satisfies the guard of a $forward$ transition, this action is not modified, and the current state moves to the target state. As an example, when action $(Ctrl1, Press, \{(On, true)\})$ is executed in Automaton A state 0, this action is forwarded, and the current state moves to state 1.
- d_0 states that if every transition guard returns $false$ when evaluating an action, then, the action is discarded, and the current state remains the same. For example, when action $(Ctrl1, Press, \{(On, true)\})$ is executed in Automaton A state 1, this action is discarded, and the current state remains in state 1.
- d_1 states that when an action satisfies the guard of a $discard$ transition, this action is discarded, and the current state moves to the target state. For instance, when action $(Ctrl1, Press, \{(On, false)\})$ is executed in Automaton A state 0, this action is discarded, and the current state moves to the same state since the transition is a self-loop.

Table 5.1: Transition rules between configurations in contract automata

$$\frac{\forall(s,\gamma,g,\alpha,s') \in T \quad g(a) = i}{(s, buf) \xrightarrow{a/a} (s, buf)} \ (f_0)$$

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

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

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

$$\frac{\exists(s,\gamma,g,\alpha,s') \in T \quad \gamma = replace \quad g(a) = true}{(s, buf) \xrightarrow{a/\alpha} (s', buf)} \ (r_1)$$

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

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

- r_1 states that when an action satisfies the guard of a $replace$ transition, this action is replaced with a replacement action and the current state moves to the target state. As an example, when action $(QualStation1, Done, \{(Pass, false), (Quality, 50)\})$ is executed in Automaton B state 1, this action is replaced with $(QualStation1, Done, \{(Pass, true), (Quality, 51)\})$ and the current state moves to state 0.
- b_1 states that when an action satisfies the guard of a $buffer$ transition, this action is added into a buffer, and the current state moves to the target state. For instance, when action $(Ctrl1, Press, \{(On, false)\})$ is executed in Automaton A state 2, this action is buffered, and the current state moves to the same state since the transition is a self-loop.
- f_2 states that a sequence of actions is released from the buffer when those actions satisfy the guards of sequential $forward$ transitions. The released

actions are executed sequentially, and the current state moves to the target state of the last transition. For example, in Automaton A state 1 an action $(Ctrl1, Press, \{(On, false)\})$ is available in the buffer because it was buffered in state 2. This action satisfies the guard $[\![(Ctrl1)Press[On = false]]\!]$ in transition 1 to 0. Therefore, the action is removed from the buffer and executed. Moreover, the current state moves from 1 to 0.

5.4 Enforcer synthesis and integration

A contract automaton specifies the modifications of application execution to satisfy certain requirements. In order to perform these modifications, we must synthesise a component of the application that can receive specific sequence of actions, and modify each action in the sequence as specified in the automaton. Therefore, a contract automaton is synthesised into an enforcer in the form of a BFB. The BFB interfaces are created to receive and send the initial and modified actions, whereas the generated ECC should be able to apply the modification of each action according to the contract automaton.

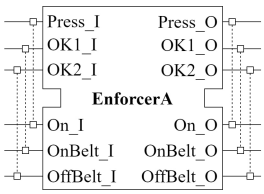
Enforcer synthesis transforms a contract automaton in Definition 5.3.1 into a BFB in Definition 2.2.3. An enforcer is denoted as $fb_e = (itf_e, DV_e, P_e, ecc_e)$. The enforcer is then integrated into the initial application $iec = (FB, EC, DC)$ to obtain a new application $iec' = (FB \cup \{fb_e\}, EC', DC')$. The enforcer synthesis is divided into two main parts: the interfaces (itf_e) and the ECC (ecc_e). The internal variables (DV_e) and the programs (P_e) are included in the ECC synthesis.

5.4.1 Synthesis of interfaces

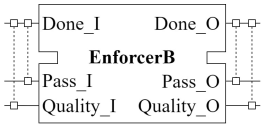
The interfaces of an enforcer are built by analysing the set of guards in the contract automaton. A pair of input and output interfaces is created for every event and data present in a guard.

Definition 5.4.1 *(Enforcer interfaces) The enforcer interfaces* $itf_e = (EI, EO, DI, DO, WI, WO)$ *are derived from the set of guards* $G = \{[\![(fb_1)e_1[B_{D_1}]]\!], [\![(fb_2)e_2[B_{D_2}]]\!], ...\}$ *in the contract automaton* (S, s^0, Γ, G, T) *such that:*

- $EI = \{e_i + \text{“\_}I\text{”}\}$,
- $EO = \{e_i + \text{“\_}O\text{”}\}$,
- $DI = \{(d \in D_i) + \text{“\_}I\text{”}\}$,



(a) Interfaces of *EnforcerA*



(b) Interfaces of *EnforcerB*

Figure 5.5: Synthesised enforcers' interfaces from Automaton A and Automaton B

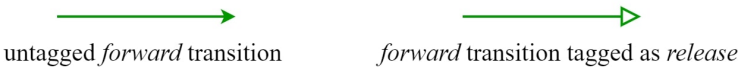


Figure 5.6: Untagged and tagged transitions

- $DO = \{(d \in D_i) + \text{“\_}O\text{”}\}$,
- $AI = \{(e_i + \text{“\_}I\text{”},\ (d \in D_i) + \text{“\_}I\text{”})\}$,
- $AO = \{(e_i + \text{“\_}O\text{”},\ (d \in D_i) + \text{“\_}O\text{”})\}$,

In the above definition, suffixes “$\_I$” and “$\_O$” are appended to the interface names to distinguish between input and output interfaces. Furthermore, prefix fb_i can be appended to the interfaces with the same name.

Example. Figure 5.5 presents the interfaces of $EnforcerA$ and $EnforcerB$ synthesised from Automaton A and Automaton B. The interfaces consist of input and output interfaces interpreted from the guards present in the contract automata. For instance, $Press\_I$, $Press\_O$, On_I, and On_O interfaces in $EnforcerA$ are synthesised from a guard $[\![(Ctrl1)Press[On = true]]\!]$ in Automaton A.

5.4.2 Synthesis of ECC

The enforcer's ECC is constructed using a set of translation patterns for transforming each transition in the contract automaton into transitions and states of ECC. However, it is necessary beforehand to identify the release of actions characterised by rule *f2* in Table 5.1 by tagging the corresponding transitions in the contract automaton. These tags help to determine which transitions in the contract automaton should be translated into ECC's transitions and states that can release the buffered actions.

Transition tags. The set of tags is defined as $\Lambda = \{u, release\}$, where u signifies an untagged transition while $release$ corresponds to transitions that can release the buffered actions. As illustrated in Figure 5.6, a transition tagged as $release$ is represented using a different endpoint style. Only $forward$ transitions can be tagged as $release$ because we should not discard nor replace the buffered actions.

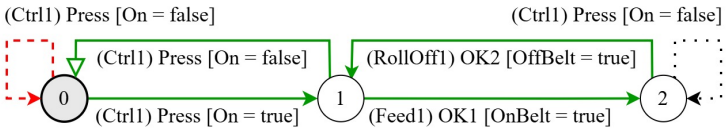


Figure 5.7: Tagged Automaton A

Definition 5.4.2 *(Tagged contract automaton)* Given a contract automaton $c = (S, s^0, \Gamma, G, T)$ and the set of tags $\Lambda = \{u, release\}$, a tagged contract automaton is $c_\Lambda = (S, s^0, \Gamma, G, \Lambda, T_\Lambda)$, such that $t = (s, \gamma, g, \alpha, \lambda, s') \in T_\Lambda$, where $\lambda \in \Lambda$. Let $Tag : T \to \Lambda$ be the function to tag every transition $t \in T$ in the automaton and $T' \subseteq T$ be the set of all transitions typed as $buffer$:

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

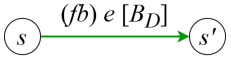
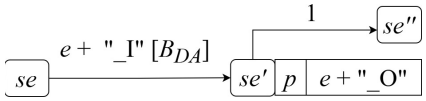
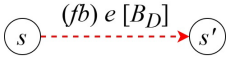
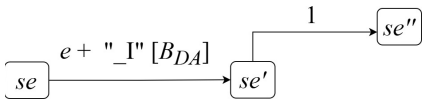
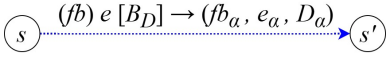
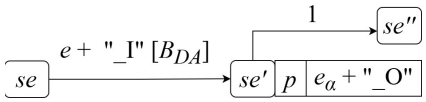
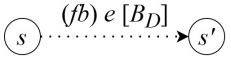
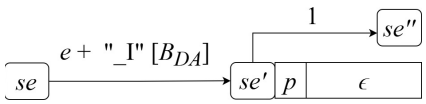
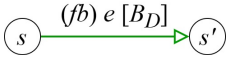
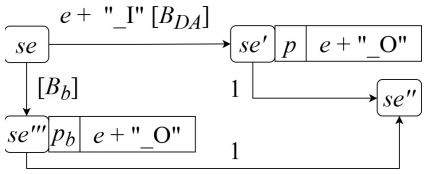
Example. Tagged Automaton A is presented in Figure 5.7. The transition from states 1 to 0 is tagged as $release$ because it is typed as $forward$ and guarded with the same guard as the transition typed as $buffer$ (the self-loop transition in state 2).

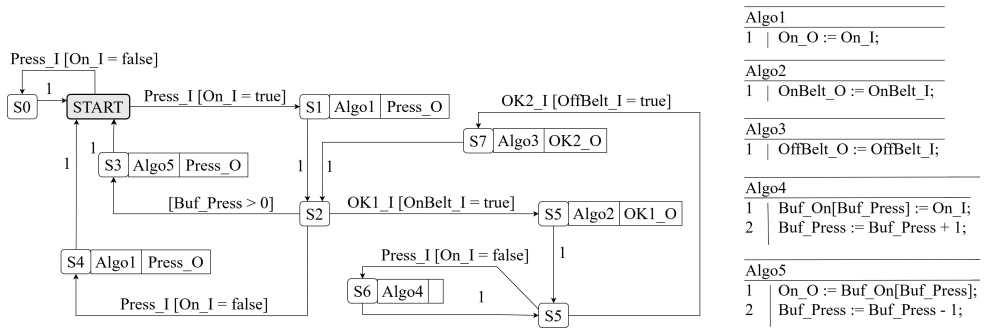
Translation patterns. A contract automaton that has been tagged can be translated into an ECC. We propose translation patterns to generate an ECC from a given contract automaton. The idea of these patterns relies on the modifications of actions represented by the contract automata transitions. For each transition type (i.e., $forward$, $discard$, $replace$, $buffer$, and $release$), we define ECC elements that can perform the modification of an action according to the contract automata transition rules (Table 5.1).

The translations patterns from contract automata transitions to elements of ECC are shown in Table 5.2. Each pattern is associated with a transition rule (e.g., $P(f_1)$ is associated with f_1). The translation patterns are described as follows:

- In $P(f_1)$, a $forward$ transition is translated to an ECC transition, leading to a state that executes the same action received by the enforcer.
- In $P(d_1)$, a $discard$ transition in a contract automaton is translated to an ECC transition, leading to a state that does not execute any action.
- In $P(r_1)$, a $replace$ transition in a contract automaton is translated to an ECC transition, leading to a state that executes the replacement action.
- In $P(b_1)$, a $buffer$ transition in a contract automaton is translated to an ECC transition, leading to a state that stores the received action in a buffer.
- In $P(f_2)$, a $release$ transition in a contract automaton is translated to two ECC transitions. The first one, se to se', leads to a state where the action is

Table 5.2: Translation patterns from contract automata to ECC transitions

Pattern	Contract automata	ECC
$P(f_1)$		
	$D = \{(x_1, y_1), (x_2, y_2), ...\}$	$DA = \{(xi_1, y_1), (xi_2, y_2), ...\}$, $p = \langle xo_1 := xi_1 \ ; \ xo_2 := xi_2 \ ; \ ...\rangle$, $xi_i = x_i + \text{“\_}I\text{”}$, $xo_i = x_i + \text{“\_}O\text{”}$
$P(d_1)$		
	$D = \{(x_1, y_1), (x_2, y_2), ...\}$	$DA = \{(xi_1, y_1), (xi_2, y_2), ...\}$, $xi_i = x_i + \text{“\_}I\text{”}$
$P(r_1)$		
	$D = \{(x_1, y_1), (x_2, y_2), ...\}$, $D_\alpha = \{(x\alpha_1, y\alpha_1), (x\alpha_2, y\alpha_2), ...\}$	$DA = \{(xi_1, y_1), (xi_2, y_2), ...\}$, $p = \langle xo_1 := y\alpha_1 \ ; \ xo_2 := y\alpha_2 \ ; \ ...\rangle$, $xi_i = x_i + \text{“\_}I\text{”}$, $xo_i = x\alpha_i + \text{“\_}O\text{”}$
$P(b_1)$		
	$D = \{(x_1, y_1), (x_2, y_2), ...\}$	$DA = \{(xi_1, y_1), (xi_2, y_2), ...\}$, $p = \langle xb_1 := xi_1 \ ; \ xb_2 := xi_2 \ ; \ ... \ ;$ $b := b + 1\rangle$, $xi_i = x_i + \text{“\_}I\text{”}$, $b = \text{“}Buf\_\text{”} + e$, $xb_i = \text{“}Buf\_\text{”} + x_i[b]$
$P(f_2)$		
	$D = \{(x_1, y_1), (x_2, y_2), ...\}$	$DA = \{(xi_1, y_1), (xi_2, y_2), ...\}$, $p = \langle xo_1 := xi_1 \ ; \ xo_2 := xi_2 \ ; \ ...\rangle$, $xi_i = x_i + \text{“\_}I\text{”}$, $xo_i = x_i + \text{“\_}O\text{”}$ $Db = \{(\text{“}Buf\_\text{”} + e, y_b)\}$, $p_b = \langle xo_1 := xb_1; xo_2 := xb_2; ...\rangle$ $xb_i = \text{“}Buf\_\text{”} + x_i[\text{“}Buf\_\text{”} + e]$



(a) ECC **(b)** ST programs

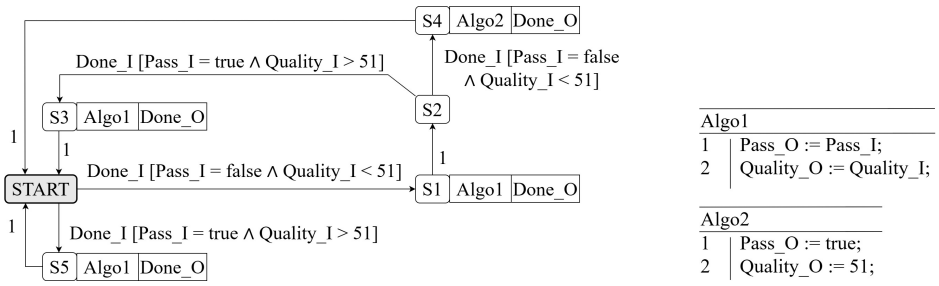
Figure 5.8: ECC and ST programs of $EnforcerA$

forwarded as in the pattern (1). The second transition, se to se''', leads to a state that releases an action from a buffer.

It is not necessary to construct a translation pattern for rule f_0 because by construction only actions specified by the guards of contract automata can be received by the enforcer. A pattern for rule d_0 is also not needed because an action is discarded by default when there exists no ECC transition guard that can be satisfied by this action. Moreover, there is an additional transition outgoing to se'' in every pattern to allow the translation of multiple transitions with the same target state.

Note that the resulting ECC may exhibit nondeterministic behaviour when multiple transitions outgoing from the same state satisfy pattern $P(f_2)$. For instance, two transitions from the same state are guarded with $[\![[Buf\_A > 0]]\!]$ and $[\![[Buf\_B > 0]]\!]$. In such a case, the ECC would arbitrarily trigger one of the transitions if both buffers contain the corresponding actions. As an alternative, the techniques could allow the user to give priorities by annotating the $buffer$ transitions to specify which one is more important to be released. However, this would make the specification process more intricate. Other possibilities are prioritising the *oldest* actions, as in the *First In, First Out* concept, or releasing the longest sequence of actions, as proposed in [FS21]. All these methods could be implemented, but they would overly complicate the translation patterns.

Example. The synthesised ECC of $EnforcerA$ from Automaton A is presented in Figure 5.8. In state $START$, there is an outgoing transition guarded with $Press\_I[ON\_I = true]$ leading to $S0$ where no action is executed (i.e., no ST program nor output event interface). This corresponds to the $discard$ transition in state 0 of the automaton. In state $S5$, there is an outgoing transition state $S6$ where interface $Press\_I$ can receive an event when the value of $On\_I = false$. There is no output event interface on $S6$, but there is an algorithm to store the information of the buffered action. Then, the unguarded transition goes back to $S5$. This loop results



(a) ECC **(b)** ST programs

Figure 5.9: ECC and ST programs of $EnforcerB$

from a self-loop $buffer$ transition in state 2 of Automaton A. The two transitions from $S2$ to $S3$ and $S4$ are synthesised according to the pattern (5) in Table 5.2. These transitions mean that the action $(Ctrl1, Press, \{(On, false)\})$ can either be released when available in the buffer (transition $S2$ to $S3$) or forwarded when received by the enforcer's input interfaces (transition $S2$ to $S4$).

The result of the translation from Automaton B to the ECC of $EnforcerB$ is depicted in Figure 5.9. From the starting state of the automaton, there are two transitions typed as $forward$. These are translated as transitions in ECC targeted to states $S1$ and $S5$, where the values of data $Pass$ and $Quality$ are forwarded by executing $Algo1$, and the event $Done$ is forwarded from interface $Done\_O$. The $replace$ transition from 1 to 0 in the contract automaton is translated to ECC's transition from $S2$ to $S4$. Here, when an action satisfies the ECC's transition guard, in state $S4$, the output event of this action is forwarded, and the data is modified by executing $Algo2$.

5.4.3 Enforcer integration

Integrating an enforcer into the application is done by disconnecting the initial connections between the interfaces and creating new connections that include the enforcer. This integration is described in Algorithm 4, which takes as input an enforcer fb_e and an IEC 61499 application iec. The algorithm generates a new application iec' where the enforcer has been integrated. The symbol $\approx$ means that two interfaces have different names only because one of them is appended with the suffix "$\_I$" or "$\_O$". For instance, the $EnforcerA$ FB interface $Press\_I$ and the $Ctrl1$ FB interface $Press$ have different names because $Press\_I$ is $Press$ appended with the suffix "$\_I$"; therefore, $Press\_I \approx Press$.

First, the enforcer is merged with the initial set of FBs (line 1). Temporary sets are initialised for facilitating the integration; EC^{new} and DC^{new} store the new

Algorithm 4: Integration of enforcer

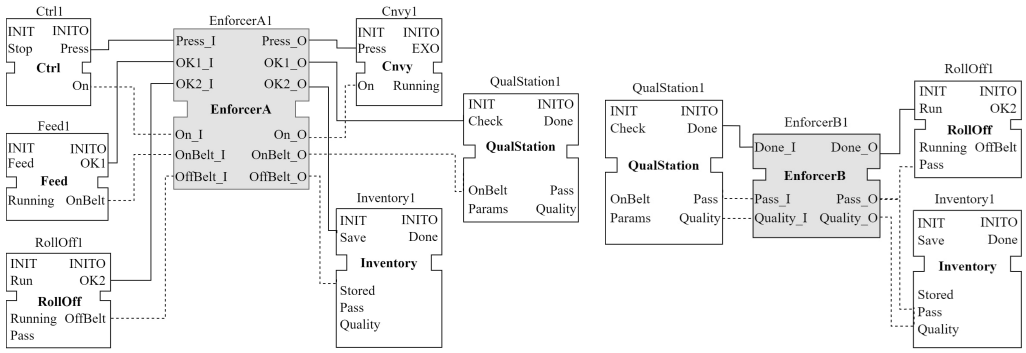
Inputs : $fb_e = (itf_e = (EI_e, EO_e, DI_e, DO_e, WI_e, WO_e), DV_e, P_e, ecc_e)$,
$iec = (FB, EC, DC)$

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

1 $FB' := FB \cup \{fb_e\}$ // enforcer is added to the set of FBs
2 $EC^{new},\ DC^{new},\ EC^{old},\ DC^{old} := \emptyset$ // temporary sets of connections
3 **foreach** $(ei, eo) \in EC$ **do**
4 **foreach** $ei_e \in EI_e$ **do**
5 **if** $eo \approx ei_e$ **then**
6 $EC^{old} := EC^{old} \cup \{(ei, eo)\}$ // an event connection to be removed
7 $EC^{new} := EC^{new} \cup \{(eo, ei_e)\}$ // an event connection to be added
8 **foreach** $eo_e \in EO_e$ **do**
9 **if** $ei \approx eo_e$ **then** $EC^{new} := EC^{new} \cup \{(eo_e, eo)\}$
10 **foreach** $(di, do) \in DC$ **do**
11 **foreach** $di_e \in DI_e$ **do**
12 **if** $do.id \approx di_e.id$ **then**
13 $DC^{old} := DC^{old} \cup \{(di, do)\}$ // data connection to be removed
14 $DC^{new} := DC^{new} \cup \{(do, di_e)\}$ // data connection to be added
15 **foreach** $do_e \in DO_e$ **do**
16 **if** $di.id \approx do_e.id$ **then** $DC^{new} := DC^{new} \cup \{(do_e, di)\}$
17 $EC' := (EC \setminus EC^{old}) \cup EC^{new}$ // new connections between interfaces
18 $DC' := (DC \setminus DC^{old}) \cup DC^{new}$

connections to be added, whereas EC^{old} and EC^{old} store the connections to be removed (line 2). Next, the algorithm iterates through the set of event connections (lines 3 to 9). When two event interfaces are associated with each other (i.e., $eo \approx ei_e$), the initial event connection is stored in EC^{old} (line 6), and the new event connections are stored in EC^{new} (lines 7 and 9). The iteration through the set of data connections in lines 10 to 16 performs the same task as in event connections. It finds and stores data connections to be removed and added in, respectively, DC^{old} and DC^{new}. Finally, the new sets of event and data connections are created in lines 17 and 18.

Algorithm 4 is correct if it results in sets of event and data interfaces that (i) contain new connections between the enforcer and the existing FBs, (ii) do not contain former connections that must be removed due to the enforcer addition, and (iii) contain former connections that need to be preserved. The first and second points hold because new and former connections are added and removed every time an interface of the enforcer is associated with the interface of an existing FB (lines 5, 10, 15 and 20). The third point holds because no connection is removed except the ones in EC^{old} and DC^{old} (lines 23 and 24).



(a) *EnforcerA* **(b)** *EnforcerB*

Figure 5.10: Enforcers integrated into the application

Example. In order to illustrate the algorithm, let us consider the integrations of *EnforcerA* and *EnforcerB* into the IEC 61499 conveyor test station. The results are depicted in Figure 5.10. Only new connections between the enforcers and the existing FBs are shown for clarity. The other connections remain the same as presented in Figure 5.1b. As described in Algorithm 4, event connections in the application associated with the enforcer's interfaces are removed, followed by the creation of new connections. For instance, to integrate *EnforcerA*, *Press* output event interface in *Ctrl1* was initially connected to *Press* input event interface in *Cnvy1*. This connection is removed, and new connections are created between *Press* in *Ctrl1* to *Press_I* and *Press_O* to *Press* in *Cnvy1*. Similarly, data connections between *Pass* in *QualStation1* to *Pass* in *RollOff1* and *Inventory1* are replaced with connections from *Pass* to *Pass_I*. Then, *Pass_O* is connected to *Pass* in both *RollOff1* and *Inventory1*.

5.5 Runtime enforcement characteristics

A runtime enforcement approach is correct when it satisfies soundness, transparency, and deadlock-freedom characteristics [RRF20; FS21; LMM23]. Soundness is satisfied when the application executes according to the requirements. Our approach must be sound because the enforcer should modify the execution of the application such that the requirements are satisfied. The runtime enforcement techniques are transparent when they do not induce unexpected behaviour. Transparency is needed because the enforcer should not make any unspecified modifications to the application execution. Deadlock-freedom is satisfied when the application can always make progress. It is required because the enforcer should not make the application stop executing. In the following sections, we explain that soundness and transparency are guaranteed

by construction. Afterwards, the use of model checking to ensure deadlock-freedom is described.

5.5.1 Soundness and transparency

Suppose that an enforcer $fb_e = (itf_e, DV_e, P_e, ecc_e)$ is synthesised from a contract automaton $c = (S, s^0, \Gamma, G, T)$. This enforcer is integrated into an initial IEC 61499 application $iec = (FB, EC, DC)$, such that $iec' = (FB \cup \{fb_e\}, EC', DC')$.

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

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

Proof (sketch). Soundness is satisfied because, by construction, the enforcer executes every possible modification of action specified in the contract automaton.

Proposition 2 states that a trace is modified if and only if the modification is specified in the contract automaton.

Proposition 2 ***(Transparency).***
$\forall \sigma \in Run(iec) : (EM^c(\sigma) = \sigma') \wedge (\sigma \neq \sigma')$ *iff* $(\sigma' \models c)$

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

5.5.2 Deadlock-freedom

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

Before deploying the modified application, the formal modelling method in Chapter 3 is first applied to generate the LTS model. Afterwards, in order to check that an LTS

model is free from deadlock, we must specify a property stating that every state has at least one successor. This property is written in MCL [MS03] as `[true*] <true> true`. The CADP model checker [Gar+13] is then used to check the property on the LTS model. If $true$ is returned, then deadlock-freedom is guaranteed.

Note that our approach for ensuring deadlock-freedom is generic. There exist other works in [Dro+16; OV21; BRS17] that can also be used to apply model checking techniques for IEC 61499 using various formal languages and model checkers.

5.6 Tool Support

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

Our implementation relies on 4DIAC-IDE [Str+08], an open-source development tool for IEC 61499 applications. The developed tools and experiments presented in this section are available in an online repository[1]. This repository also contains more examples for readers interested in them. One of them is a more complex conveyor test station containing 23 composite FBs with more than 100 FBs inside of them in total. There is also a temperature control system with an enforcer that consists of 32 states and 44 transitions.

The technical detail for synthesising enforcers is presented in Figure 5.11. A program written in the Java programming language has been developed to implement the synthesis techniques described in Section 5.4. It returns an enforcer (i.e., a basic FB) in the form of an XML file for a given contract automaton, which is written in a text file. 4DIAC-IDE is used to read the generated enforcers. Every new enforcer must be exported and compiled together with the runtime environment called FORTE [2]. Once all the steps are completed, the enforcer can be integrated into the application.

Figure 5.12 shows the steps required to use our approach for adapting physical systems. The first step is to deploy the application. 4DIAC-IDE provides a comprehensive guide about the deployment of IEC 61499 applications [4DI24]. Once the application is running, a contract automaton is specified to be synthesised as an

[1] `https://gitlab.inria.fr/ifaqriza/runtime-enforcement-for-iec-61499`

[2] This compilation process is detailed in the official webpage of 4DIAC-IDE: `https://www.eclipse.org/4diac/en_help.php?helppage=html/installation/install.html`

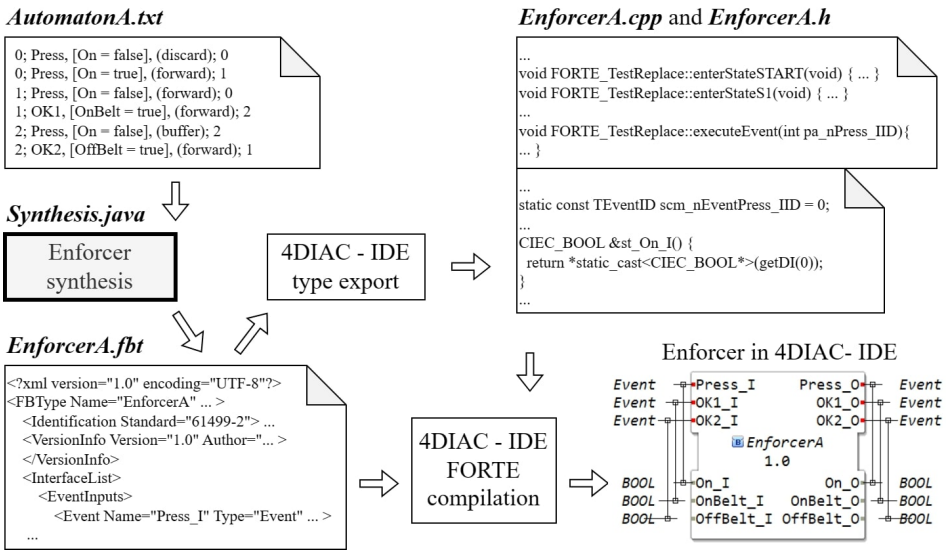


Figure 5.11: Implementation of enforcer synthesis

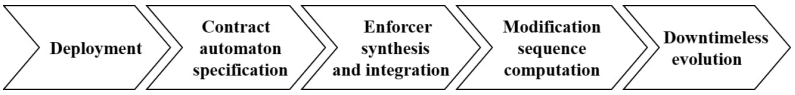


Figure 5.12: Adaptation of physical systems using runtime enforcement for IEC 61499

enforcer. Algorithm 4 computes a target application in which the enforcer has been integrated. Next, a modification sequence is computed from the initial application (without an enforcer) and the target application. This sequence is a list of operations required for preserving dependencies during the dynamic modification (e.g., the connections of an FB must be removed before deleting that FB). The work in [PS21] describes the automated computation of the modification sequence. Finally, the initial application is modified into the target application according to the modification sequence without stopping the physical system. Existing tools and frameworks, such as the one proposed in [PRG20], can be used to complete this task.

5.7 Experimental results

The goal of this experiment is to compare the performance of IEC 61499 applications before and after the integration of enforcers. This is done to make sure that the enforcers do not induce significant performance overhead. We take several real-world examples of IEC 61499 applications from the existing literature. Requirements are then specified using contract automata. These automata are synthesised as enforcers and integrated into the corresponding applications. Finally, we execute

Table 5.3: Experimental results

Application		CA		ECC		Itf.		ET(ms)	
Name	FBs	St.	Tr.	St.	Tr.	Ev.	Da.	Be.	Af.
1. Convey. 1A [ZL14]	7	3	6	10	14	6	6	59167	59108
2. Convey. 1B [ZL14]	7	2	4	6	7	2	4	59167	59151
3. Convey. 2A [ZL14]	23	3	6	10	14	6	6	83106	83115
4. Convey. 2C [ZL14]	23	10	20	30	40	2	4	83106	83129
5. Temp. A [ZL14]	4	2	4	6	8	2	2	10005	10003
6. Temp. B [ZL14]	4	5	12	18	24	2	2	10005	10004
7. Temp. 2C [ZL14]	8	11	22	32	44	2	2	10003	10005
8. Capping A [ZSE13]	5	2	4	6	8	4	0	10005	10004
9. Capping B [ZSE13]	5	5	12	18	24	4	0	10005	10003
10. Blink A [IDE24]	3	2	2	4	4	4	0	10002	10003
11. Blink B [IDE24]	3	2	4	6	8	4	0	10002	10005
Average								32233	32230

multiple simulations of these applications to obtain the average execution time with and without enforcers. These simulations are performed using 4diac IDE [Str+08] and FORTE runtime environment.

The results of our experiments are presented in Table 5.3. The first two columns contain the application names and the numbers of FBs representing the size of the application. The next two columns contain the number of states and transitions in the contract automata to represent the complexity of each contract automaton. The fifth and sixth columns contain the number of states and transitions of the enforcers' ECCs, whereas the seventh and eighth columns contain the number of event and data interfaces of the enforcers. They represent the complexity of the synthesised enforcer. The last two columns contain the average execution time before and after enforcers' integrations in milliseconds. Note that some of the experiments were performed using the same applications but different contract automata. Therefore, they are named differently (e.g., applications Blink A and Blink B).

The results in Table 5.3 show the applicability of our approach in terms of performance. In most cases, there is no significant overhead when an enforcer is integrated into the application. Experiment 4 resulted in the highest increase of execution time

(23 ms). This is because, in this specific experiment, multiple complex enforcers were integrated into an application that consists of many FBs. However, the increase is much less than a second, thus negligible. Therefore, we may conclude that our approach is applicable in terms of performance.

Threats to Validity. The experimental results are obtained using simulations in 4diac IDE. There are two potential threats to the validity of these results. The first one concerns the time intervals between the events in the simulations that may not reflect those in physical systems. In physical systems, some events are triggered due to the interactions between the sensors and the environment. In our simulations, these events were created using additional FBs that can trigger events periodically called E_CYCLE. Nevertheless, there should not be a significant threat to the validity of the results in this regard because the execution time of the enforcer does not depend on the execution time of other FBs. Secondly, our experiments do not consider distributed applications. In such applications, there may be additional overheads when the enforcers must receive and send events through the network [Lin+15]. Dedicated frameworks and extensions for distributed IEC 61499 applications, such as the one proposed in [BWZ23], can be used to mitigate the communication delays that may arise in such cases. Overall, this aspect does not pose a major threat to the validity of the results.

5.8 Concluding remarks

This chapter provides a solution to **RQ2** described in Chapter 1. The runtime enforcement concept can be applied to compute IEC 61499 applications that satisfy some given requirements. The method starts with an expressive and compact input specification called contract automaton, which allows the user to specify different modifications of actions according to the requirements. An action can be forwarded, discarded, replaced, or buffered. This automaton is transformed into an enforcer in the form of a BFB. Afterwards, the enforcer is integrated into the application to modify its runtime execution as specified in the requirements.

The approach is demonstrated using a real-world IEC 61499 conveyor test station. We showed that our runtime enforcement method is sound and transparent. A tool is developed to automate the synthesis of enforcers. Furthermore, experiments were presented to show that the enforcer does not induce significant performance overhead.

6

Guided evolution of IEC 61499 applications

IEC 61499 applications may need to be evolved during their life cycles 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. To avoid these risks, it is important to define a systematic and reliable software evolution method. This chapter offers a solution that generates evolution guidelines, which can bridge the gap between the requirements and the target application. Conventionally, the developers must consider several possible modifications when there are requirements that need to be satisfied. The guidelines generated by our techniques can help them to make necessary changes to the application without introducing erroneous behaviour and extra complexity.

Our approach is illustrated in Figure 6.1. It takes as input (1) an initial IEC 61499 application and (2) the 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 [ZL14]. Such applications are characterised by the presence of 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.

This chapter is structured as follows. Section 6.1 introduces an extended version of the IEC 61499 behavioural model. Section 6.2 presents a mixing tank system as the case study for illustrating our approach. Section 6.3 explains the guided evolution techniques. Section 6.4 shows a prototype of the approach implementation. Section 6.5 concludes.

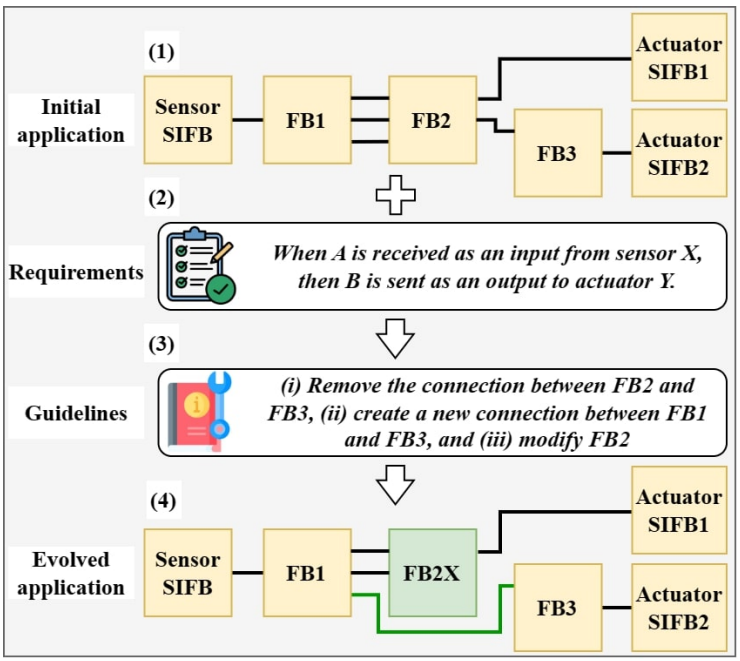


Figure 6.1: Overview of the approach

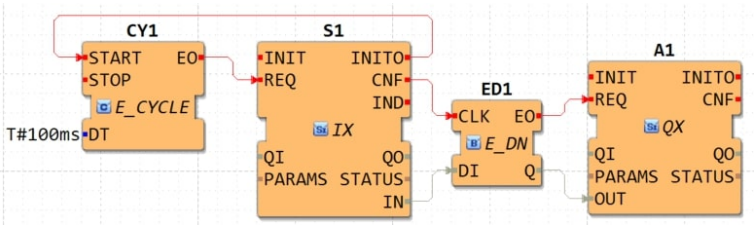
6.1 Extended behavioural model

The behavioural model used in this chapter is adapted from the LTS model introduced in Chapter 3. The notion of action is extended so that it consists of an output and a set of inputs. An output is composed of interfaces that trigger an event and data, whereas an input consists of interfaces that receive these events and data. The purpose of this extension is to distinguish between outputs that are sent from the sensor SIFBs and inputs that are received by the actuators SIFBs.

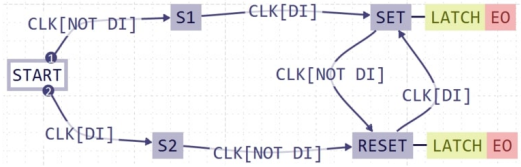
Definition 6.1.1 (Extended action) *An extended action is a tuple* (o, I)*, where:*

- $o = (fb, eo, DO)$ *is an output consisting of a source FB* fb*, an event interface* eo*, and a set of data* DO*,*
- $I = \{(fb'_1, ei_1, DI_1), (fb'_2, ei_2, DI_2), ..., (fb'_n, ei_n, DI_n)\}$ *is a set of inputs, each consists of a target FB* fb'*, an event interface* ei*, and a set of data* DI*.*

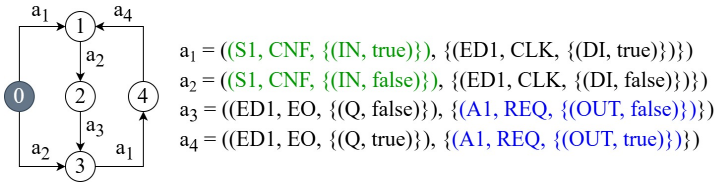
Example. To illustrate the extended model, an application is presented in Figure 6.2a. $CY1$ is an FB that generates events periodically. $S1$ and $A1$ are sensor and actuator SIFBs. $ED1$ is a BFB with an ECC shown in Figure 6.2b. This BFB regulates the flow of events in the application to make the actuator SIFB only receive an event whenever the sensor SIFB's data value changes. Suppose $S1$ triggers an event from CNF with $IN = true$. Then, we can write this as an action



(a) Application



(b) ECC of $ED1$



(c) Extended LTS

Figure 6.2: Example of IEC 61499 application with SIFBs and its extended LTS

$((S1, CNF, \{(IN, true)\}), \{(ED1, CLK, \{(DI, true)\})\})$. Figure 6.2c shows the extended LTS. The highlighted texts are the outputs (green) and inputs (blue) of the sensor and actuator SIFBs. The model shows both behaviour and structure of the application. 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).

6.2 Case study

A mixing tank system [CC19] in Figure 6.3 is used in this chapter 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 6.1 (Behaviour 1). This behaviour is a standard mixing process in which both pumps start or stop pumping when the tank is empty or full. The liquid is drained when the tank is full, and the

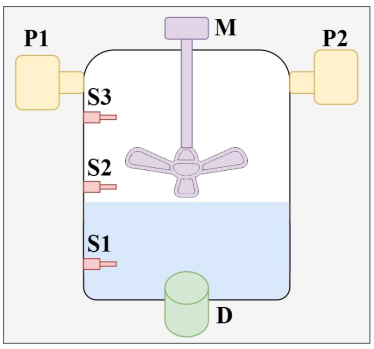


Figure 6.3: Mixing tank system

Table 6.1: 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.

mixer is on when there is enough liquid. The other two behaviours in the table are the target behaviours of the 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 6.2: 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.

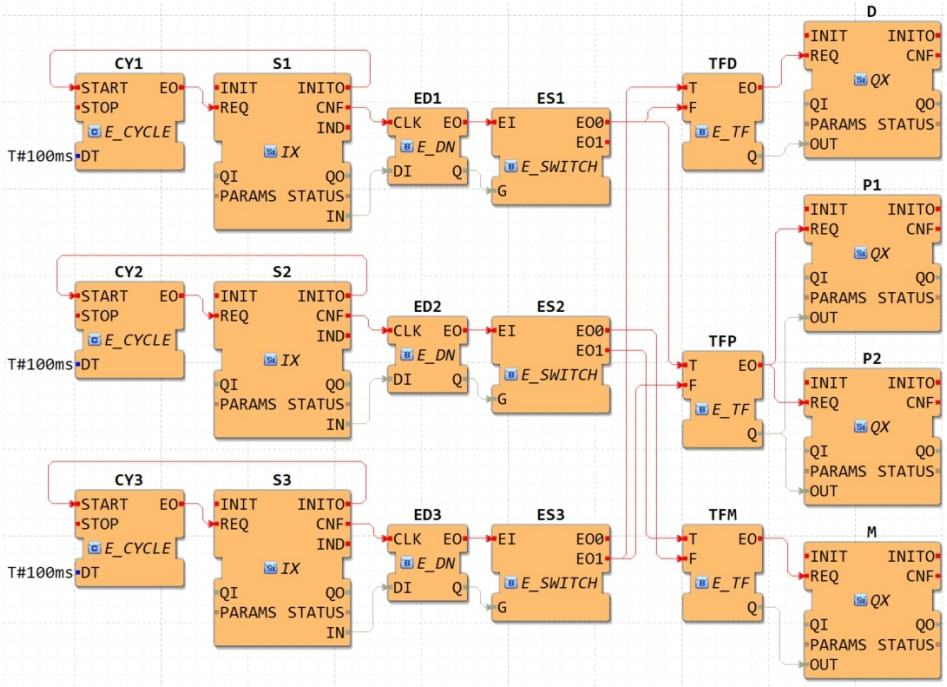


Figure 6.4: Initial IEC 61499 application of the mixing tank system

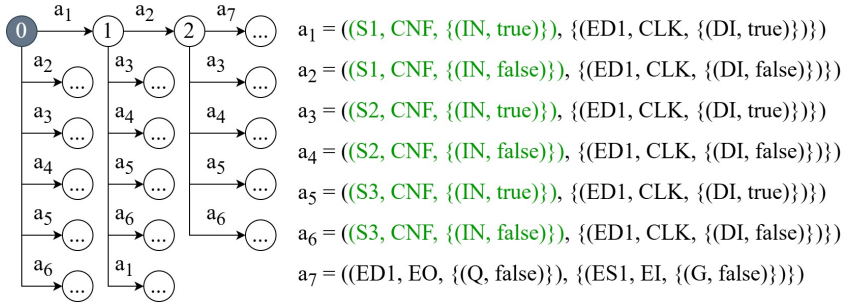


Figure 6.5: Fragment of the initial application model

The IEC 61499 application describing the initial behaviour of the mixing tank is depicted in Figure 6.4. *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 6.2. The other FBs (e.g., *ED1*, *ES1*, and *TFD*) regulate the interactions between the SIFBs such that the mixing tank operates according to Behaviour 1.

We use the translation techniques described in Chapter 3 on the application in Figure 6.4 to obtain a model consisting of 49392 states and 340956 transitions. To illustrate, we present a fragment of this model in Figure 6.5. It consists of three states and their outgoing transitions. The model shows that the initial state can trigger six possible actions, which are triggered by the sensor SIFBs. Furthermore,

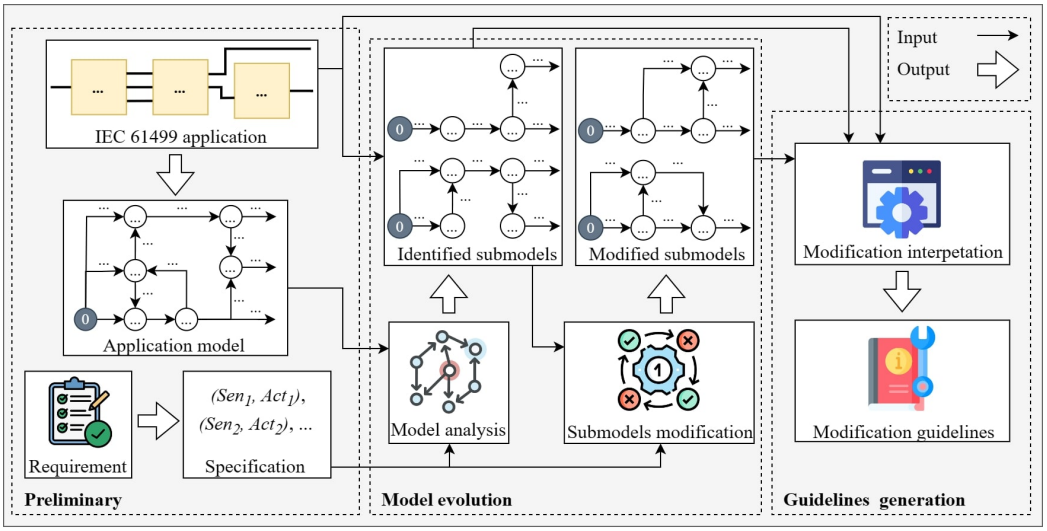


Figure 6.6: Overview of the guided evolution techniques

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).

The following section explains how the guidelines for evolving the application in Figure 6.4 into applications that execute according to Behaviour A and Behaviour B can be automatically generated.

6.3 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 method, which consists of three modules, is presented in Figure 6.6. In the *Preliminary* module, the application is translated into an LTS model, 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.

6.3.1 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 an LTS model, whereas the requirements are interpreted as a specification.

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 6.3.1 (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 6.1 can be represented as interactions using the descriptions in Table 6.2. For instance, the first statement in Behaviour A is represented as (Sen, Act), where $Sen = (S1, CNF, \{(IN, true)\}),\ (S1, CNF, \{(IN, false)\})$ and $Act = \{(P1, REQ, \{(OUT, true)\}), (D, REQ, \{(OUT, false)\})\}$.

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 in the application's initial behaviour. This set is called a specification of the requirements.

Definition 6.3.2 (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 6.1. Afterwards, we use the description of the sensor and actuator SIFBs in Table 6.2 to translate the requirements into a specification.

Example. Table 6.3 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

Table 6.3: Mixing tank system specifications according to Behaviour A and Behaviour B

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

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 in this specification $(P2, REQ, \{(OUT, true)\}) \notin Act_1$ because P2 should not start pumping when the liquid moves below S1.

6.3.2 Model evolution

The *Model evolution* module generates submodels that can guide the application's evolution. The initial model describes all possible sequences of actions 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 5 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 action sequences that begin with the actions triggered by the sensor SIFBs and end with the actions received by the actuator SIFBs. Function $isNext$ checks if the output of an action is the next one in the sequence, whereas function *isTrig* checks if an action is triggered by the FB to which the previous action was sent. Function $addToModel$ adds a transition into a

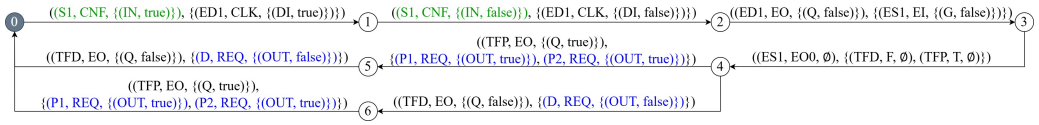
Algorithm 5: Model analysis

input : $m_{init} = (S, s_0, A, T),\ r = \{(Sen_1, Act_1), (Sen_2, Act_2), ..., (Sen_n, Act_n)\}$

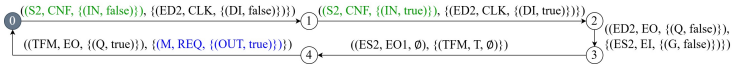
output : $M_{ident} = \{m_1, m_2, ..., m_n\}$

1 **foreach** $(Sen, Act) \in r$ **do** // iterate through the set of interactions
2 $m' := (\varnothing, s_0, \varnothing, \varnothing)$ // initialise a new submodel
3 $identify\ (s_0, m_{init}, m', Sen, \epsilon)$
4 $M_{ident} := M_{ident} \cup m'$ // add the submodel to the set of identified submodels

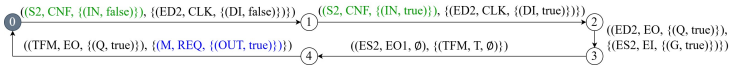
5 **identify** (s_c, m, m', Sen, a_p)
6 **let** T_c be the set of transitions outgoing from s_c **in**
7 **foreach** $t = (s, a, s') \in T_c$, **do** // iterate through the set of transitions
8 **if** $isNext\ (a, Sen) \vee isTrig\ (a, a_p)$ **then** // check if the transition is relevant
9 $addToModel\ (m', t)$ // add the transition to the submodel
10 $identify\ (s', m, m', Sen, a)$



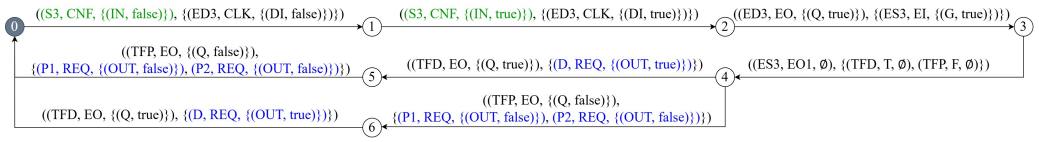
(a) The first identified submodel according to Requirement A (Submodel A1)



(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)

Figure 6.7: Identified submodels

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 action's output is the next one in Sen or triggered by the FB to which the previous action was sent (lines 8 and 9).

Example. We apply Algorithm 5 on the initial model of the application to obtain submodels shown in Figure 6.7. There are two submodels for each specification

because both Specification A and Specification B contain two interactions. Each submodel describes the sequences of actions that can be triggered according to the sequence of outputs in an interaction. For instance, in submodel A1 (Figure 6.7a), the first two transitions correspond to the sequence of outputs in Sen_1 (see Table 6.3, Specification A). This submodel describes the sequences of actions 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.

Algorithm 6: 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 {(m1,(Sen1,Act1)),(m2,(Sen2,Act2)),
     ...,(mn,(Senn,Actn))} such that mi ∈ Mident and (Seni,Acti) ∈ r in
       /* iterate through the set of pairs                                  */
2      foreach (m = (S, s0, A, T),(Sen, Act)) ∈ P do
         /* initialise the set of inputs to be removed from the submodel    */
3        Irem := getInputs(m) \ Act
         /* initialise the set of outputs to be added to the submodel       */
4        Iadd := Act \ getInputs(m)
5        remInputs (s0, m, Irem)
6        addInputs (s0, m, Iadd)
7        Mmodif := Mmodif ∪ m

8  remInputs (sc, m, Irem)
9      let Tc be the set of transitions outgoing from sc in
10       foreach t = (s, a, s') ∈ Tc do
11         Itmp := a.I ∩ Irem
           /* check if the transition is labelled with an input to be removed */
12         if checkRem(Itmp, m) then removeTrans (t, m)
13         else modifyTrans (Itmp, t, m)
14         remInputs (s', m, Irem)

15 addInputs (sc, m, Iadd)
16     let Tc be the set of transitions outgoing from sc in
17       foreach t ∈ Tc do
           /* build interleaving transitions                                */
18         if checkAdd(t) then interleave (Iadd, t, m)
19         addInputs (s', m, Iadd)
```

Submodels modification. Algorithm 6 describes how the identified submodels are modified according to the specification. It takes the set of identified submodels M_{ident} and the 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 $(M, REQ, \{(OUT, true)\})$.

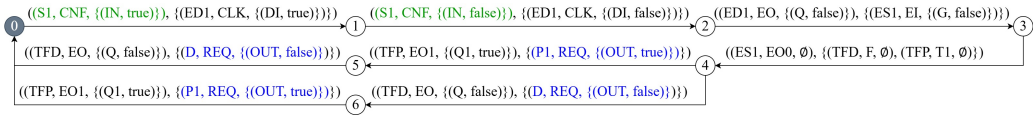
The algorithm iterates through the set of pairs to modify the identified submodels (lines 2 to 7). In each iteration, it initiates the sets of inputs to be removed and inputs to be added (lines 2 and 3). Function $remInputs$ traverses a submodel to find transitions labelled with actions containing inputs that need removal (lines 8 to 16). If the action 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).

Example. Figure 6.8 shows the modified submodels obtained using Algorithm 6. Each submodel now contains sequences of actions that satisfy the specification. For instance, the input $(P2, REQ, \{(OUT, true)\})$ is removed in the modified submodel A1 (Figure 6.8a) 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 Figure 6.8b) because it is one of the expected inputs in Act_2.

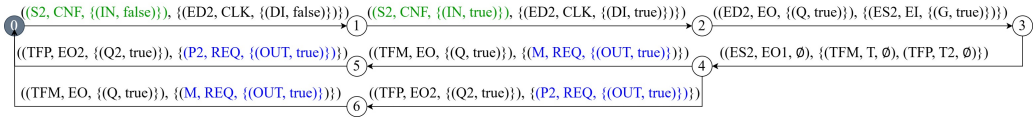
6.3.3 Guidelines generation

The *Guidelines generation* module analyses the initial and modified submodels to generate evolution guidelines. The idea is to find the differences between the transitions in the initial submodel and the transitions in the modified submodel.

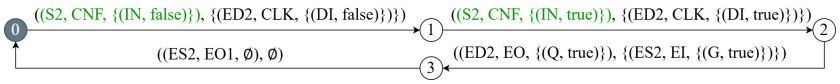
Algorithm 7 describes the guidelines generation. It takes an identified submodel and a modified submodel as input. The algorithm returns event connections to be removed, event connections to be added, data connections to be removed, data connections to be added, and BFB to be modified. Function $pref : T \rightarrow 2^{T^*}$ is used to obtain sequences of transitions preceding a given transition.



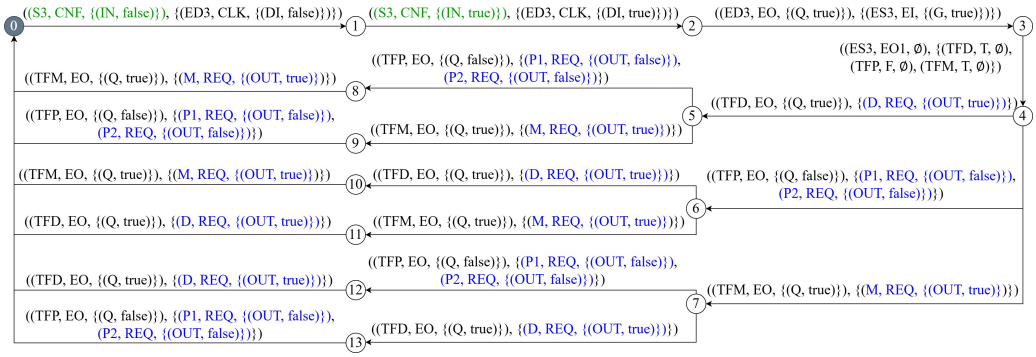
(a) Modified A1



(b) Modified A2



(c) Modified B1



(d) Modified B2

Figure 6.8: Modified submodels

The algorithm iterates through the set of transitions in the identified and modified submodels (lines 1 and 2). Afterwards, it checks if the transitions are preceded by the same sequences and have the same outputs on their labels (line 3). The inputs of the transitions are then compared (lines 4 to 11). This comparison results in one of the following three possible outcomes. Firstly, connections between FBs must be removed if an input is missing in the modified transition (lines 4 to 6). Secondly, an FB must be modified if there is a new input in the modified transition and a new transition exists in one of the subsequent transitions (lines 7 to 8). Lastly, new connections must be added if a new input exists in the modified transition (lines 9 to 11).

Example. The evolution guidelines to obtain applications with Behaviour A and Behaviour B are presented in Table 6.4 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 Figures 6.7 and 6.8. For instance, in Guidelines B (i), the connection between

Algorithm 7: Guidelines generation

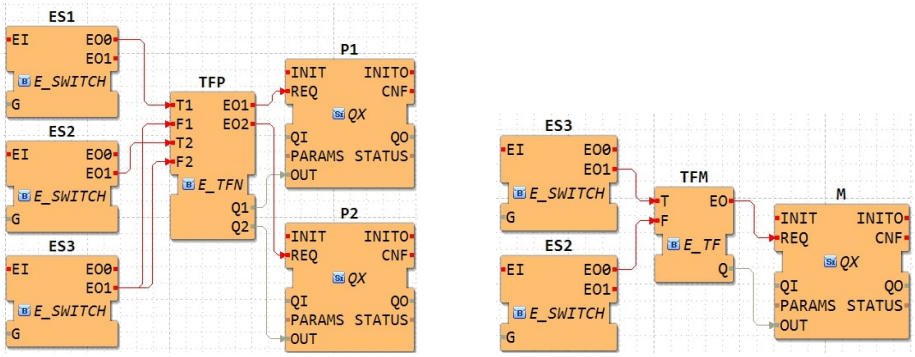
input : $m_{ident} = (S_i, s_i^0, A_i, T_i),\ m_{modif} = (S_m, s_m^0, A_m, T_m)$
output : $EC_{rem},\ EC_{add},\ DC_{rem},\ DC_{add},\ FB_{mod}$

```
/* iterate through the set of transitions in the initial submodel */
foreach t_i = (s_i, (o_i = (fb, eo, DO), I_i), s'_i) ∈ T_i do
    foreach t_m = (s_m, (o_m, I_m), s'_m) ∈ T_m do
        /* compare with each transition in the modified submodel */
        if (pref(t_i) = pref(t_m)) ∧ (o_i = o_m) then
            /* remove connections if an input is removed from the corresponding
               transition */
            if ∃ i_i = (fb', ei, DI) ∈ I_i : i_i ∉ I_m then
                EC_rem := EC_rem ∪ {(eo, ei)}
                DC_rem := DC_rem ∪ (DO, DI)
            /* modify the BFB if there is a transition in the modified submodel
               that is not in the initial one */
            else if
              (∃ i_m = (fb', ei, DI) ∉ I_m : i_m ∈ I_i) ∧ (∃ t'_m = (s'_m, a, s''_m) ∈ T_m : t'_m ∉ T_i)
              then
                FB_mod := EC_add ∪ {(eo, ei)}
            /* add new connections if an input is added to the corresponding
               transition */
            else if (∃ i_m = (fb', ei, DI) ∉ I_m : i_m ∈ I_i) then
                EC_add := EC_add ∪ {(eo, ei)}
                DC_add := DC_add ∪ (DO, DI)
```

Table 6.4: 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)

$EO1$ 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



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

Figure 6.9: Fragments of the evolved applications

fragments of the evolved applications according to the guidelines are shown in Figure 6.9.

6.4 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 application model. The second one is the specification with the same format as in Table. 6.3. It generates the identified and modified submodels in textual format as output. Every module in our approach can be automated. However, associating the input and output SIFBs with physical activities requires manual intervention (e.g., Table 6.2). Nevertheless, this is not an issue because SIFBs include documentation explaining the meaning of their input and output interfaces.

Figure 6.10 shows a fragment 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 fragment shows the identified and modified submodels corresponding to the first interaction of Requirement B (see Figures 6.7c and 6.8c).

6.5 Concluding remarks

This chapter presents an alternative solution to **RQ2**. We make use of various algorithms and analysis techniques on the behavioural model of the application to generate guidelines for evolving the application. The method begins with identifying

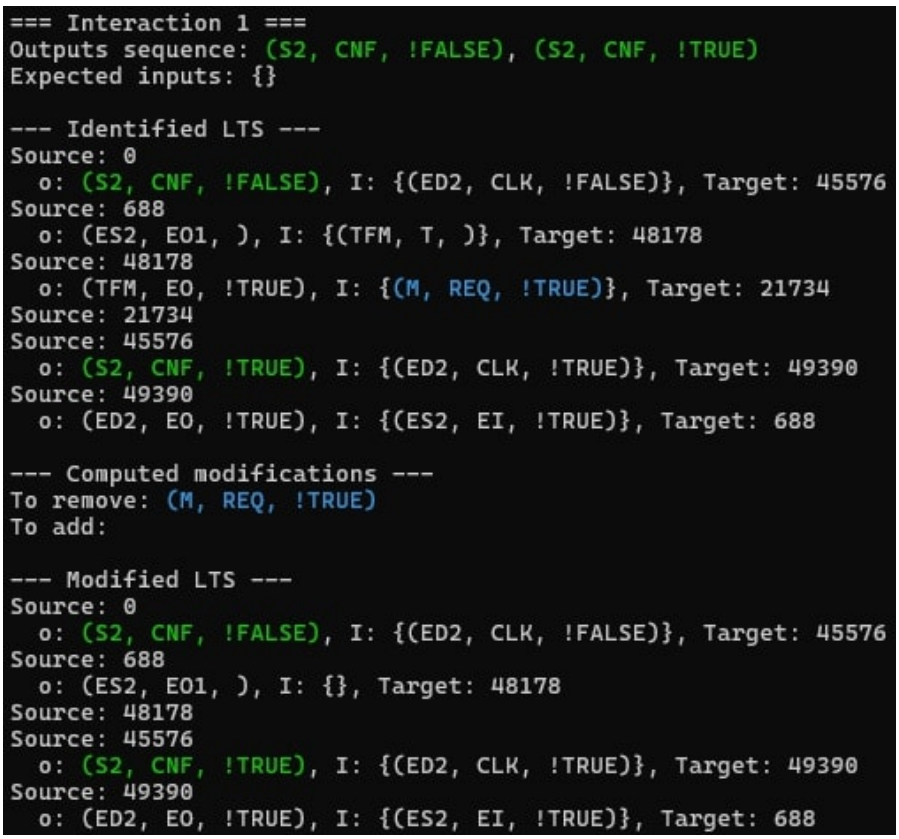


Figure 6.10: Fragment of the prototype's outputs

the parts of the model that are relevant to the requirements. Afterwards, these submodels are analysed and modified according to the requirements. Submodels before and after modifications are compared to generate evolution guidelines. These guidelines help to make the necessary modifications without adding extra complexity or creating erroneous behaviour. The approach is demonstrated on an IEC 61499 mixing station. A prototype has been developed to automate the analysis and modification of submodels.

7

Related work

This chapter surveys related works and is structured as follows. Section 7.1 compares our formal modelling techniques with existing works that propose to model the behaviour of IEC 61499 applications. Section 7.2 compares our probabilistic model checking approach with similar verification techniques. Section 7.3 compares our runtime enforcement and guided evolution techniques with research works aimed at evolving automation systems to satisfy certain requirements. Section 7.4 concludes.

7.1 Formal modelling

Petri nets. The authors in [VH99] propose one of the earliest modelling techniques for IEC 61499 applications when the standard was still a draft and called IEC 1499. An application is modelled as a type of Petri net [Rei85] extended with condition and event arcs [RH95]. A model consists of places, transitions, and arcs. Every arc between a place and a transition is labelled with either a condition (e.g., an ST program is completed) or an event (e.g., an input event is received). Petri net is chosen to avoid state space explosion [Cla+11]. Its non-interleaving semantics permit firing several transitions simultaneously, which makes it suitable for representing an event sent to multiple FBs. In our LTS model, an event triggered by a source FB and events received by target FBs is represented as a single action, which helps to mitigate the number of states. Moreover, FB's internal executions, such as the completion of an ST program, are not included in our model because they are not closely related to the system's physical behaviour. Overall, LTS provides a more compact representation of the application behaviour by taking into account the sequences of actions. Furthermore, an LTS can easily be extended into a PTS by enriching the transitions with probabilistic values because we consider events and data that can be monitored (unlike the executions of ST programs that are encapsulated inside BFBs).

SMV languange. The authors in [PDV15b; PDV15c] propose to generate models in the SMV language [Cim+99] from IEC 61499 basic function blocks. A BFB is translated into an SMV module consisting of variable declarations, guard definitions, states, and transitions. This translation is formalised using Abstract State Machine (ASM) [Gur93] and implemented as *fb2smv* [Dmi24] tool. In [Dro+21], this modelling approach is extended with timing aspects to support the verification of real-time properties. The proposed translation methods are comparable with our translation patterns. Their approach is more straightforward because the syntax of SMV allows them to explicitly declare state variables and $next$ functions to represent ECC states and transitions. In our approach, we use (sub)processes and process calls. However, their works only describe the translations of individual BFBs into SMV modules, which means that modelling the entire application behaviour is not yet supported. The issue is that modules in SMV can not be used in a parallel composition with synchronisation on some actions, as in LNT. Moreover, SMV does not support the addition of probabilistic values on its transitions.

Promela. The work in [SV21] proposes to model IEC 61499 applications with Promela [Neu14], which is the input language for the SPIN model checker [Hol97]. The modelling approach relies on the ECC's Operation State Machine (OSM). Three Promela processes are created to represent the three states in OSM. Each process contains conditional statements on the events and data followed by *goto* statements to determine the next OSM state whenever a condition is satisfied. The authors use the *message channels* in Promela to represent the transmissions of events between the FBs so that the model of the whole application can be created. In comparison, we use the parallel composition construct in LNT to incorporate all the processes translated from the FBs in the application. Moreover, this work also does not produce models that can be extended with probabilistic values because SPIN does not support probabilistic model checking.

Execution traces. Formal modelling of IEC 61499 execution traces is proposed in [Lia+22]. A trace is defined as a sequence of *system states* consisting of timestamps and updates. An update contains any observable changes in the application at runtime, such as an ECC's current state moving to a new one. In total, there are 11 possible types of updates. The authors propose this notion of execution trace for observation purposes. Thus, every detail in the application's progression at runtime is included. In comparison, our execution trace is dedicated to enriching the behavioural model with the likelihood of execution sequences. Execution details such as timestamps are not necessary because LTS considers only events and data that the application can trigger.

Misc. In [DVH06], IEC 61499 applications are modelled as Prolog programs [CM94]. The modelling begins with the creation of a Prolog *term* for each ECC state in the application. Afterwards, a tree-like structure is built to represent the global state of the application. Finally, a Prolog *production rule* is generated to represent the input and output of each ECC transition. The case study shows that the model can help answer questions such as "At which values of X will the ECC of FB Y be in state Z?". In summary, the model proposed in their work is useful for reachability analysis and checking the implication of data values in the application's ECCs and ST programs. In comparison, our model focuses on a higher level of abstraction, which is the sequences of events and data of the application. This allows us to verify properties that are closely related to the system's physical activities.

In [PDV15a], IEC 61499 applications are modelled as programs in Ptolemy II [Pto14], which is an embedded system development framework with an actor-oriented design paradigm [LNW03]. FBs are transformed into actors, and each of their ECCs is translated into an Extended State Machine (ESM), which defines the execution logic of an actor. This translation makes use of an algorithm for converting Moore machines (ECC) to Mealy machines (ESM). The modelling techniques proposed in this work and ours are aimed at different objectives. Their main goal is to create computational models that can leverage the loosely defined execution semantics of IEC 61499. This is done by mapping the standard's syntactic components to Ptolemy II. In contrast, our focus is to model the likelihood of the application's executions for probabilistic model checking. To achieve this goal, we generate LTS models that can be extended into PTSs to check some given probabilistic properties using the CADP model checker [Gar+13].

In [KS21], the authors define a transformation of IEC 61499 applications to Business Process Model Notation (BPMN) [AK15] models. This transformation makes use of translation patterns from ECC components to BPMN notations. For instance, an ECC state is translated into a BPMN task. A BPMN model represents an automation system from a business perspective and enables quantitative analysis of process models. This analysis assumes that choices in the process have equal probability. In comparison, we enrich the transitions in our probabilistic models with probabilistic values computed from the monitored traces to improve the quantitative analysis.

7.2 Verification

Symbolic model checking. In [XPV21], a model checking tool-chain is developed and used to verify an IEC 61499 drilling station. The authors use a simulation model developed using NxtStudio to represent the drilling station. The tool-chain relies on (i) the *fb2smv* [Dmi24] tool for converting the application's FBs into SMV models and (ii) the NuSMV model checker [Cim+99]. The verification results show that the approach can check safety properties, such as the table should not rotate while the drilling process is ongoing. Our probabilistic model checking was also applied to a drilling station system. However, we focus on verifying the quantitative aspects of the system, such as the likelihood of detecting a certain type of material. Another difference is that their model checker is not applied to the model of the whole application because the *fb2smv* only supports the translation of individual FBs. Therefore, certain FBs that need to be verified must be determined before applying the approach. In comparison, we use the CADP probabilistic model checker on the PTS, which represents the probabilistic behaviour of the entire drilling station. Furthermore, our experiments were conducted on a physical system, whereas theirs were executed on a simulation model.

The work in [OV21] introduces a technique based on model checking to visually explain properties' violations. As in the aforementioned works, the approach begins with automatic translations of IEC 61499 FBs into SMV models using the *fb2smv* tool. Then, the NuSMV model checker is used to generate counterexamples from the models and some given properties. Finally, the counterexamples are utilised to infer influence paths in a graphical interface. These paths are presented visually to the users to help them debug IEC 61499 applications. In this approach, the visual explanations are aimed at debugging violations of safety properties. In contrast, we apply PMC to several probabilistic models (computed from the same application but with different traces) and properties. This results in the probabilities of action sequences that can be presented visually using charts to analyse the quantitative aspects of the system, such as productivity.

Statistical model checking. The works in [SAM23; AMO23; BH21] apply statistical model checking [Leg+19] (SMC) to analyse and optimise industrial systems in various aspects. In [SAM23], SMC is combined with the fault tree analysis method to evaluate the probabilities of system failure and power consumption. The authors in [AMO23] use system modelling language (SysML) to model cyber-physical systems. The case study on artificial pancreas shows that the approach can verify critical safety properties. Finally, SMC is used to analyse the performance of production

lines in [BH21]. These works employ SMC, while ours uses PMC. SMC is known to have a lower memory requirement, but it returns approximative results as output. In contrast, PMC may require exploring the entire state space but provides more accurate results.

Runtime verification. The authors in [Jhu+21] propose a method for monitoring adapter connections in IEC 61499 applications. The monitor contains state machines specifying certain properties. When a property is violated, an event is triggered as a notification. The work in [Tra+20] also relies on runtime verification. The authors propose to integrate *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. The probabilistic model checking approach in our work also involves monitoring the application. However, the monitor does not verify properties but records the execution traces to be used for computing probabilistic models. Furthermore, these runtime verification works are also comparable to our runtime enforcement approach. The difference is that the added FB in our approach modifies the runtime execution according to some given requirements.

The work in [YR10] uses Esterel for verifying the safety properties of IEC 61499 applications. One of the main features of this work is the use of *synchronous observers*, where monitoring FBs are integrated to check the application's correctness at runtime. A case study on the cruise control system shows that the approach is able to verify safety properties, such as the *cruise mode* should deactivate when the brake pedal is pressed. This work focuses on verifying safety properties at runtime. In contrast, we monitor the application execution when running and use the monitored traces to infer probabilistic values in the model. Their verification method returns whether the property is satisfied or not, whereas we inform the user about the likelihood of the property being satisfied.

Misc. The work in [PVS12] presents the verification of an IEC 61499 pick-and-place system. The authors make use of IEC 61499 to Petri net modelling techniques [VH99] and the SESA model checker [KH11]. The system is developed according to *controller-client* architecture, and the properties are specified to check the interactions between controllers and workers. For instance, the approach can verify that when controller cylinders descend, the client cylinders should stand still. In this work, properties are specified to express the expected interactions between system components. In comparison, probabilistic properties in our approach are specified to express the likelihood of execution sequences.

A case study on the verification of an IEC 61499 airflow control system is presented in [SG04]. The application is modelled as a network of timed automata. The UPPAAL model checker [LPY97] is used to verify predefined properties on the model. These properties correspond to the timing aspects of the application. For instance, a property ensures that the output relative to a sampling event is returned before a new sampling event is triggered. This work aims to verify the application's correctness, taking into account the timing aspects. In comparison, we focus on the quantitative aspects of the application.

In [Bor+05], model checking is applied to verify the correctness of a turntable system, which resembles the drilling station. Three model checkers were tested: SPIN [Hol97], CADP [Gar+13], and UPPAAL [LPY97]. The system, written in χ language, is translated to the model checkers' specification languages. Properties such as deadlock freedom and fairness were checked. This work does not consider the presence of the environment, such as workers interacting with the system. On the contrary, our approach considers the environment using probabilistic models computed from execution traces.

7.3 Evolution

Downtimeless evolution. The authors in [Hum+06] 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 [SFV06; SVZ13] propose verification techniques to evaluate the correctness of the reconfiguration process. Their method is extended in [SZ11] for distributed applications. The work in [PS21] generates the correct order of the reconfiguration operations (i.e., *RECONF*) using dependency graphs. This approach is refined in [PHS22a] to support real-time systems by considering the timing constraints and applying the priority ceiling protocol. Finally, the work in [PHS22b] describes the techniques to generate rollback sequences when reconfigurations of applications need to be cancelled. This collection of works (i.e., [Hum+06; SFV06; SVZ13; SZ11; PS21; PHS22a; PHS22b]) facilitates the seamless evolution of IEC 61499 applications without stopping their executions. In comparison, our proposed methods focus on finding the target application for a given initial application and evolution requirements. Runtime enforcement is applied to satisfy the requirements by modifying the runtime execution with enforcers, whereas the guided evolution techniques could help by generating step-by-step guidelines. Our methods can be integrated

into the aforementioned downtimeless evolution techniques to build an end-to-end framework for evolving IEC 61499 applications.

Feature-oriented evolution. The work in [Hin+18] proposes a featured-oriented evolution method for industrial automation systems. It relies on the mapping between features and codes. The evolution is simplified by automatically updating the code when a new feature is introduced. This approach is useful for adding and removing features. In comparison, both our guided evolution and runtime enforcement techniques help to modify existing features according to the requirements. For instance, their approach can be used when there is a new tester component that needs to be added to the IEC 61499 conveyor test station, whereas ours can be used when the behaviour of the existing tester needs to be modified.

Reconfiguration techniques. The work in [SJC14] 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 associations 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, whereas we deal with the behavioural aspect of the application. For example, their approach can evolve an application by adding an FB in a control device because the requirements specify that this type of FB must exist in every control device. In comparison, both our guided evolution and runtime enforcement techniques can add an FB into the application because this FB can make the application produce the sequences of events and data specified in the requirements.

Design patterns. The authors in [WSZ20; Son+21; Son+22] propose to support the adaptability of IEC 61499 applications using specific design patterns. A distributed hierarchical design pattern is studied in [WSZ20]. This pattern can be used when the application involves many parallelism. In [Son+21], this pattern is extended to further support the reconfigurability and reusability by differentiating FBs based on their interactions. The most recent one [Son+22] introduces new FBs to the pattern dedicated to error detection and handling. These works help the users in building adaptable IEC 61499 applications using specific design patterns, which are applied during the development phase (i.e., before deployment). Their objective is to simplify the application's modification when it needs to be evolved according to some requirements at runtime. In contrast, our methods can be applied after the deployment phase and can help the evolution process itself by either integrating an

enforcer to change the runtime execution or generating guidelines for modifying the application.

Self-adaptive systems. The authors in [Ade+22] propose to support adaptive automation systems by defining the formal notion of *explanation*. Explanations contain information about certain adaptation decisions. Each explanation is characterised by content, effect, and cost. Probabilistic model checking is used to obtain the optimal explanation, which allows human operators to understand the change in system behaviours during an adaptation process. When a questionable adaptation occurs, the operator may cancel the process. Their notion of explanations can be compared with the notion of guidelines in our guided evolution techniques. The main difference is that explanations can only be obtained whenever the adaptation has been determined by the system's maintainers. In comparison, guidelines can be used to determine the correct modifications that need to be done during the adaptation itself.

The work in [CR23] introduces a goal modelling notation called EDGE to support self-adaptive systems. This notation is proposed based on five *desiderata* (i.e., requirements), such as allowing automated goal selection. EDGE models are synthesised into goal controllers using the PRISM model checker [KNP02]. The goal controllers in this work are comparable with enforcers in our runtime enforcement approach. However, a goal controller aims to change the system's objective. For instance, it can make the conveyor test station perform a new task, such as repairing defective materials. In comparison, an enforcer focuses on changing the system's behaviour, such as making the conveyor test station accept every material. Alternatively, we can also use guided evolution techniques to generate the modifications required to make the application satisfy the requirements.

Runtime enforcement. The authors in [Pin+17] propose runtime enforcement techniques for cyber-physical systems, in particular, the heart pacemaker system. The requirements are expressed as Synchronous Discrete Time Automata (SDTA). This language allows, for instance, to forbid two different pace signals (i.e., *atrial* and *ventricular* paces) to happen simultaneously. In comparison with our work, the states of SDTA are distinguished into two types: accepting and non-accepting. A contract automaton does not require this distinction because it aims only at specifying how the system can achieve a desired behaviour (as opposed to undesired behaviour where non-accepting states are necessary). In SDTA, when the transition goes to a non-accepting state, the corresponding action is replaced with another existing action on a transition outgoing from the same source state. This is similar to our *replace* mechanism, where the original and replacement actions are specified on the

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

The work in [LMM23] focuses on the application of runtime enforcement for industrial automation systems relying on the standard IEC 61131-3 [IEC02]. The controller programs are expressed using a language based on a timed process calculus, while the correctness of these programs is specified using time correctness properties. Similarly to ours, a property is synthesised as an enforcer. The properties that can be specified in this work are specific, such as *bounded eventually*, *bounded absence*, and *bounded maximum duration*. In comparison, we propose to express properties using contract automata. This allows the users to specify requirements to be enforced by using different types of transitions. Moreover, their work targets the IEC 61131-3 standard, where programs execute using a scan-based model, whereas ours focuses on IEC 61499, which relies on an event-driven execution model.

Specification languages. Security Automata (SA), introduced in [Sch00], is one of the earliest runtime enforcement specification languages. It is an FSM-based language that runs in parallel with the system. A security automaton can specify to halt the system when a transition corresponding to a certain action is not available in the current state. Halting is not an option in our case because the automation system must continue making progress. For instance, the industrial materials would be congested if the conveyor test station is halted. In comparison, our contract automata can specify to discard, replace, or buffer the undesired action without stopping the IEC 61499 application.

Edit Automata (EA) described in [LBW05] is a well-known language to enforce system properties. It extends SA's capability to insert and suppress actions. This feature is called buffering in contract automata. In addition to that, contract automata can discard or replace actions. The discard mechanism can be useful when it is not necessary to buffer certain actions. For instance, when a conveyor is switched on, discarding the action that corresponds to switching on the conveyor is preferred over buffering the action.

A specification language based on temporal logic [Pnu77] called Event-Driven Temporal Logic (EDTL) is introduced in [Zyu+21]. The language can specify requirements based on six unique attributes. For instance, the *trigger* attribute

describes the starting event of the requirement, while the *reaction* attribute is associated with the ending event. In comparison, we use different types of transition in contract automata to specify the desired application behaviour.

7.4 Concluding remarks

In this chapter, we have presented existing works related to our contributions. The comparisons between these works and ours show that our approaches differ in several ways.

Existing works on formal modelling techniques for IEC 61499 do not take into account the runtime execution of the application. In comparison, we use monitoring techniques to enrich our model with probabilistic values to represent the likelihood of execution sequences. Also, most of the works focus on generating models for each individual FB (except modelling with Promela), whereas our method can be applied to the entire application. Furthermore, the models in our approach represent the system's behaviour at the application level, which is unlike most other works that consider the internal executions of FBs. The advantage of concentrating on this level of abstraction is that we can specify properties that are closely related to the physical activities of the system.

The existing verification methods for IEC 61499 mostly apply conventional model checking to verify safety properties. There are also other existing techniques, such as runtime verification. These methods return a verdict for the satisfaction of the property. In comparison, our approach can produce the likelihood of the property satisfaction. Overall, our PMC approach provides an alternative for users who are interested in verifying the application's quantitative aspects, such as productivity.

Various research works exist to help evolve IEC 61499 applications. Some of these works focus on the evolution process from an initial application to a new application without stopping the execution. However, these works do not describe the computation to find the new target application that can satisfy the requirements. Two works that propose to generate target applications aim to add features and reconfigurations of control devices. To the best of our knowledge, a method for computing target applications based on the behavioural aspects has not been proposed. There are also design patterns and self-adaptive concepts proposed to support evolving industrial automation systems. However, these proposals must be applied before the deployment. Our runtime enforcement and guided evolution techniques help to fill these gaps. They can both be used to obtain a new application that satisfies

some given requirements. These requirements can be specified according to the desired behaviour using contract automata or specifications. Finally, we can use our approach to satisfy requirements given at runtime.

8

Conclusion

This chapter concludes the thesis. It consists of a summary of the contributions and perspectives for the future work.

8.1 Summary of contributions

Industrial automation has been gaining potential in recent years due to the advancement in information technology. However, it faces many challenges that must be addressed to optimise its capability. This thesis focuses on two major challenges. The first one is how to consider probabilistic behaviour originating from the environment to verify and analyse the system's quantitative aspects. The second challenge is how to evolve the system according to some given requirements at runtime. The contributions of this thesis consist of several methods that are proposed to deal with these challenges. These methods are aimed at systems designed using IEC 61499, which is an emerging industrial automation standard.

Probabilistic Model Checking (PMC) techniques are proposed to verify and analyse the quantitative aspects of IEC 61499 applications that are influenced by nondeterministic environments. This approach involves both static and dynamic analyses, which allows for capturing both the design time and runtime behaviours of the system. At design time, we propose modelling techniques to generate a Labelled Transition System (LTS) model from a given application. An LTS consists of states and transitions labelled with events and data triggered by the application. At runtime, we monitor the application to obtain execution traces. A trace is then used to extend an LTS into a Probabilistic Transition System (PTS) model by computing probabilistic values on the model's transitions. The CADP model checker can be used to check some probabilistic properties on the generated PTS. Experiments were conducted on an IEC 61499 drilling station. The results show that this method permits us to analyse the impact of the environment towards the quantitative aspects of the system, such as productivity.

IEC 61499 comes with a downtimeless evolution feature, which can make an application evolve without stopping the execution. Our contributions focus on finding the target application according to some given requirements. For this, we propose two different approaches. The first one implements the runtime enforcement concept to IEC 61499 by integrating enforcers to make the application execution follow the requirements. This approach begins with specifying requirements as contract automaton, which describes the modifications of events and data using types of transition. An automaton is then synthesised as an enforcer, which is then integrated to modify the execution of the application according to the initial requirements. The second approach uses the application's behavioural models to generate evolution guidelines. It starts with identifying the parts of the model that are relevant to the requirements. These parts are then analysed and modified so that the sequences of events and data in the model follow the desired requirements. The models are then compared to generate evolution guidelines. These two approaches allow users to specify some requirements and obtain either an application with enforcers for modifying the runtime execution or guidelines that bridge the gap between the requirements and the target application.

8.2 Future work

Probabilistic model checking. The main perspective for the IEC 61499 PMC approach is to consider other external aspects originating from the environment, such as network communications between control devices in distributed applications. The objective is to consider the possibility that an event sent through the network may be lost during transmission. This possibility is not taken into account in the current probabilistic model because we assume that events are always successfully transmitted. Several methods could be applied to consider this network communication aspect. One of them is to extend the model with transitions that represent event losses. The execution traces must also contain relevant information in order to compute the probabilistic values on these transitions. For this, the monitoring FB must be revised to recognise whether an event is lost during transmission.

A secondary perspective is to create a framework that can apply the PMC approach at runtime. This could be done by improving the functionality of the monitoring FB. More precisely, this FB is enhanced with functions to compute the probabilistic model and to verify some given properties. This would allow users to analyse the impact of the environment towards the system's quantitative aspects during its execution.

Runtime enforcement. The main perspective for runtime enforcement techniques is to consider distributed applications. In such applications, FBs are mapped to different control devices, and communication delays may occur whenever events are sent through these devices' networks. To mitigate these delays, we must decide which control device is suitable for the enforcer whenever it needs to be integrated into the application. A possible solution is to map the enforcer to a control device where most of the corresponding FBs are located. Furthermore, in distributed applications, the transmission of an event may fail when the network is congested. This means that an enforcer that is supposed to send or receive such an event would also fail to apply the necessary modifications as specified in the contract automaton. Overall, further studies must be done to apply the runtime enforcement techniques for distributed IEC 61499 applications.

Guided evolution. Possible future work for the guided evolution techniques is to integrate the approach with existing tools and frameworks for evolving IEC 61499 applications without stopping their execution. This would allow seamless evolutions of IEC 61499 applications, in which developers would only need to specify requirements when evolving applications. The first step to achieve this goal is to apply the modifications specified in the generated guidelines to the application. This requires an algorithm that can modify the application according to the guidelines. Afterwards, the modified application is used as input for a specific downtimeless evolution framework so that the running application can be modified seamlessly to satisfy the requirements.

Bibliography

[4DI24] 4DIAC. *Deployment of IEC 61499 Applications*. `https://fordiac.sourceforge.net/ehelp/html/overview/deployment.html`. 2024 (cit. on p. 62).

[AK15] Gustav Aagesen and John Krogstie. "BPMN 2.0 for Modeling Business Processes". In: *Handbook on Business Process Management 1, Introduction, Methods, and Information Systems, 2nd Ed*. Springer, 2015, pp. 219–250 (cit. on p. 85).

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996 (cit. on p. 19).

[Ade+22] Sridhar Adepu, Nianyu Li, Eunsuk Kang, and David Garlan. "Modeling and Analysis of Explanation for Secure Industrial Control Systems". In: *ACM Trans. Auton. Adapt. Syst.* 17.3-4 (2022), pp. 1–26 (cit. on p. 90).

[Alm+11] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. "An Overview of Formal Methods Tools and Techniques". In: *Rigorous Software Development: An Introduction to Program Verification*. London: Springer London, 2011, pp. 15–44 (cit. on p. 2).

[AMO23] Abdel-Latif Alshalalfah, Otmane Ait Mohamed, and Samir Ouchani. "A framework for modeling and analyzing cyber-physical systems using statistical model checking". In: *Internet of Things* 22 (2023), p. 100732 (cit. on p. 86).

[ANS83] ANSI/IEEE. "IEEE Standard Glossary of Software Engineering Terminology". In: *ANSI/IEEE Std 729-1983* (1983), pp. 1–40 (cit. on p. 3).

[Ant20] Tom Mejer Antonsen. *PLC Controls with Structured Text (ST), V3 Monochrome: IEC 61131-3 and best practice ST programming*. Books on Demand, 2020 (cit. on pp. 11, 13).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cit. on pp. 3, 15).

[BH21] Paolo Ballarini and András Horváth. "Performance Analysis of Production Lines Through Statistical Model Checking". In: *Proc. of EPEW'21, and ASMTA'21*. Vol. 13104. LNCS. Springer, 2021, pp. 264–281 (cit. on pp. 86, 87).

[BRS17] Zeeshan Ejaz Bhatti, Partha S. Roop, and Roopak Sinha. "Unified Functional Safety Assessment of Industrial Automation Systems". In: *IEEE Trans. Ind. Informatics* 13.1 (2017), pp. 17–26 (cit. on pp. 3, 62).

[Bor+05] Elena M. Bortnik, Nikola Trcka, Anton Wijs, et al. “Analyzing a *chi* model of a turntable system using SPIN, CADP and UPPAAL”. In: *J. Log. Algebraic Methods Program.* 65.2 (2005), pp. 51–104 (cit. on p. 88).

[BWZ23] Friederike Bruns, Bianca Wiesmayr, and Alois Zoitl. “Supporting Model-Based Network Specification for Time-Critical Distributed Control Systems in IEC 61499”. In: *Proc. of CASE'23*. IEEE, 2023, pp. 1–7 (cit. on p. 65).

[CR23] Radu Calinescu and Genaína Nunes Rodrigues. “Goal Controller Synthesis for Self-Adaptive Systems”. In: *Proc. of FormaliSE'23*. IEEE, 2023, pp. 1–6 (cit. on p. 90).

[Cha+24] David Champelovier, Xavier Clerc, Hubert Garavel, et al. “Reference Manual of the LNT to LOTOS Translator (Version 7.3)”. 155 pages. 2024 (cit. on pp. 6, 15).

[CC19] Tim Chen and C.Y.J. Cheng. “Modelling and verification of an automatic controller for a water treatment mixing tank”. In: *Desalination and Water Treatment* 159 (2019), pp. 318–326 (cit. on p. 69).

[Cim+99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. “NUSMV: A New Symbolic Model Verifier”. In: *Proc. of CAV'99*. Vol. 1633. LNCS. Springer, 1999, pp. 495–499 (cit. on pp. 84, 86).

[Cla+11] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. “Model Checking and the State Explosion Problem”. In: *Proc. of LASER'11*. Vol. 7682. LNCS. Springer, 2011, pp. 1–30 (cit. on p. 83).

[CM94] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (4. ed.)* Springer, 1994 (cit. on p. 85).

[DA11] Ashley De Sa and Sarim Al Zubaidy. “Gas turbine performance at varying ambient temperature”. In: *Applied Thermal Engineering* 31.14 (2011), pp. 2735–2739 (cit. on p. 3).

[Dmi24] Dmitrii Drozdov. *IEC 61499 Function blocks XML code to SMV converter*. `https://github.com/dmitrydrozdov/fb2smv`. 2024 (cit. on pp. 84, 86).

[Dro+21] Dmitrii Drozdov, Victor Dubinin, Sandeep Patil, and Valeriy Vyatkin. “A Formal Model of IEC 61499-Based Industrial Automation Architecture Supporting Time-Aware Computations”. In: *IEEE Open Journal of the Ind. Electronics Society* 2 (2021), pp. 169–183 (cit. on pp. 11, 84).

[Dro+16] Dmitrii Drozdov, Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. “Formal verification of cyber-physical automation systems modelled with timed block diagrams”. In: *Proc. of ISIE'16*. IEEE, 2016, pp. 316–321 (cit. on pp. 3, 62).

[DV08] Victor Dubinin and Valeriy Vyatkin. “On Definition of a Formal Model for IEC 61499 Function Blocks”. In: *EURASIP J. Embed. Syst.* 2008 (2008) (cit. on pp. 12, 22).

[DVH06] Victor Dubinin, Valeriy Vyatkin, and Hans-Michael Hanisch. “Modelling and Verification of IEC 61499 Applications using Prolog”. In: *Proc. of ETFA’06*. IEEE, 2006, pp. 774–781 (cit. on p. 85).

[Fal10] Yliès Falcone. “You Should Better Enforce Than Verify”. In: *Proc. of RV’10*. Vol. 6418. LNCS. Springer, 2010, pp. 89–105 (cit. on pp. 5, 17).

[Fal+18] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. “Runtime Failure Prevention and Reaction”. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*. Vol. 10457. LNCS. Springer, 2018, pp. 103–134 (cit. on pp. 5, 17).

[FS21] Yliès Falcone and Gwen Salaün. “Runtime Enforcement with Reordering, Healing, and Suppression”. In: *Proc. of SEFM’21*. Vol. 13085. LNCS. Springer, 2021, pp. 47–65 (cit. on pp. 57, 60).

[GBP20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. “The 2020 Expert Survey on Formal Methods”. In: *Proc. of FMICS’20*. Vol. 12327. LNCS. Springer, 2020, pp. 3–69 (cit. on pp. 2, 14).

[Gar+13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. “CADP 2011: a toolbox for the construction and analysis of distributed processes”. In: *Int. J. Softw. Tools Technol. Transf.* 15.2 (2013), pp. 89–107 (cit. on pp. 6, 15, 29, 36, 62, 85, 88).

[Gli07] Martin Glinz. “On Non-Functional Requirements”. In: *Proc. of RE’07*. IEEE Computer Society, 2007, pp. 21–26 (cit. on p. 3).

[Gua10] Massimo Guarnieri. “The Roots of Automation Before Mechatronics [Historical]”. In: *IEEE Industrial Electronics Magazine* 4.2 (2010), pp. 42–43 (cit. on p. 1).

[Gur93] Yuri Gurevich. “Evolving algebras 1993: Lipari guide”. In: *Specification and validation methods*. Ed. by Egon Börger. Oxford University Press, 1993, pp. 9–36 (cit. on p. 84).

[Hin+18] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, et al. “Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems”. In: *Proc. of ETFA’18*. IEEE, 2018, pp. 107–114 (cit. on p. 89).

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295 (cit. on pp. 84, 88).

[Hum+06] Oliver Hummer, Christoph Sünder, Alois Zoitl, et al. “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 (cit. on pp. 2, 88).

[IDE24] 4diac IDE. *4diac IDE Documentation*. `https://eclipse.dev/4diac/en_help.php`. 2024 (cit. on pp. 19, 64).

[IEC12] IEC. “International Electrotechnical Commission, Functional blocks - Part 1: Architecture, 2nd edn, IEC 61499-1”. In: *IEC Geneva* (2012) (cit. on pp. 1, 2, 9).

[IEC02] IEC. “Programmable controllers-part 3: Programming languages”. In: *IEC 61131-3 (Ed. 2.0)* (2002) (cit. on pp. 1, 9, 91).

[ISO89] ISO. *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. Tech. rep. 8807. ISO, 1989 (cit. on p. 15).

[Jhu+21] Pranay Jhunjhunwala, Jan Olaf Blech, Alois Zoitl, Udayanto Dwi Atmojo, and Valeriy Vyatkin. “A Design Pattern for Monitoring Adapter Connections in IEC 61499”. In: *Proc. of ICIT’21*. IEEE, 2021, pp. 967–972 (cit. on p. 87).

[Kel76] Robert M. Keller. “Formal Verification of Parallel Programs”. In: *Commun. ACM* 19.7 (1976), pp. 371–384 (cit. on pp. 6, 16, 21).

[KH11] Mohamed Khalgui and Hans-Michael Hanisch. “Automatic NCES-based specification and SESA-based verification of feasible control components in benchmark production systems”. In: *Int. J. Model. Identif. Control.* 12.3 (2011), pp. 223–243 (cit. on p. 87).

[KS21] Ajay Krishna and Gwen Salaün. “Business Process Models for Analysis of Industrial IoT Applications”. In: *IoT ’21: 11th Int. Conf. on the IoT*. ACM, 2021, pp. 102–109 (cit. on p. 85).

[Kro+19] Lene Kromann, Nikolaj Malchow-Møller, Jan Rose Skaksen, and Anders Sørensen. “Automation and productivity—a cross-country, cross-industry comparison”. In: *Industrial and Corporate Change* 29.2 (July 2019), pp. 265–287 (cit. on p. 1).

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM: Probabilistic Symbolic Model Checker”. In: *Proc. of TOOLS’02*. Vol. 2324. LNCS. Springer, 2002, pp. 200–204 (cit. on pp. 15, 90).

[KNP18] Marta Kwiatkowska, Gethin Norman, and David Parker. “Probabilistic Model Checking: Advances and Applications”. In: *Formal System Verification: State-of the-Art and Future Trends*. Springer, 2018, pp. 73–121 (cit. on pp. 4, 15).

[LMM23] Ruggero Lanotte, Massimo Merro, and Andrei Munteanu. “Industrial Control Systems Security via Runtime Enforcement”. In: *ACM Trans. Priv. Secur.* 26.1 (2023), 4:1–4:41 (cit. on pp. 60, 91).

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a Nutshell”. In: *Int. J. Softw. Tools Technol. Transf.* 1.1-2 (1997), pp. 134–152 (cit. on p. 88).

[LS91] Kim Guldstrand Larsen and Arne Skou. “Bisimulation through Probabilistic Testing”. In: *Inf. Comput.* 94.1 (1991), pp. 1–28 (cit. on p. 35).

[LNW03] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. “Actor-Oriented Design of Embedded Hardware and Software Systems”. In: *J. Circuits Syst. Comput.* 12.3 (2003), pp. 231–260 (cit. on p. 85).

[Leg+19] Axel Legay, Anna Lukina, Louis-Marie Traonouez, et al. “Statistical Model Checking”. In: *Comput. and Softw. Science - State of the Art and Perspect.* Vol. 10000. LNCS. Springer, 2019, pp. 478–504 (cit. on p. 86).

[Ley92] F. Ley. “Programmable logic controllers - Architecture and applications : Gilles Michel”. In: *Autom.* 28.3 (1992), pp. 652–653 (cit. on p. 2).

[Li+18] Qing Li, Qianlin Tang, Iotong Chan, et al. “Smart manufacturing standardization: Architectures, reference models and standards framework”. In: *Comput. Ind.* 101 (2018), pp. 91–106 (cit. on p. 1).

[Lia+22] Tatiana Liakh, Radimir Sorokin, Daniil Akifev, Sandeep Patil, and Valeriy Vyatkin. “Formal model of IEC 61499 execution trace in FBME IDE”. In: *Proc. of INDIN’22*. IEEE, 2022, pp. 588–593 (cit. on p. 84).

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. “Edit automata: enforcement mechanisms for run-time security policies”. In: *Int. J. Inf. Sec.* 4.1-2 (2005), pp. 2–16 (cit. on pp. 48, 91).

[Lin+15] Per Lindgren, Johan Eriksson, Marcus Lindner, et al. “Response time for IEC 61499 over Ethernet”. In: *Proc. of INDIN’15*. IEEE, 2015, pp. 1206–1212 (cit. on p. 65).

[MR18] Radu Mateescu and José Ignacio Requeno. “On-the-fly model checking for extended action-based probabilistic operators”. In: *Int. J. Softw. Tools Technol. Transf.* 20.5 (2018), pp. 563–587 (cit. on p. 16).

[MS03] Radu Mateescu and Mihaela Sighireanu. “Efficient on-the-fly model-checking for regular alternation-free mu-calculus”. In: *Sci. Comput. Program.* 46.3 (2003), pp. 255–281 (cit. on p. 62).

[MR14] B.R. Mehta and Y.J. Reddy. *Industrial Process Automation Systems: Design and Implementation*. Elsevier Science, 2014 (cit. on p. 2).

[MBB17] Orna Muller, Ayelet Butman, and Moshe Butman. “Opening a (Sliding) Window to Advanced Topics”. In: *Proc. of ITiCSE’17*. ACM, 2017, pp. 52–57 (cit. on p. 38).

[Neu14] René Neumann. “Promela Formalization”. In: *Arch. Formal Proofs* 2014 (2014) (cit. on p. 84).

[ORe17] Gerard O’Regan. *Concise Guide to Formal Methods - Theory, Fundamentals and Industry Applications*. Undergraduate Topics in Computer Science. Springer, 2017 (cit. on pp. 2, 14).

[OV21] Polina Ovsiannikova and Valeriy Vyatkin. “Towards user-friendly model checking of IEC 61499 systems with counterexample explanation”. In: *Proc. of ETFA’21*. 2021, pp. 01–04 (cit. on pp. 3, 62, 86).

[PDV15a] Cheng Pang, Wenbin William Dai, and Valeriy Vyatkin. “Towards IEC 61499 models of computation in Ptolemy II”. In: *Proc. of IECON’15*. IEEE, 2015, pp. 1988–1993 (cit. on p. 85).

[Pan+14] Cheng Pang, Sandeep Patil, Chen-Wei Yang, Valeriy Vyatkin, and Anatoly Shalyto. “A portability study of IEC 61499: Semantics and tools”. In: *Proc. of INDIN’14*. IEEE, 2014, pp. 440–445 (cit. on p. 2).

[Par+23] A. Parant, D. Zander, F. Gellot, and A. Philippot. “IEC 61499 Control Architectures Evaluation for Automation Software Development”. In: 56.2 (2023). 22nd IFAC World Congress, pp. 3660–3665 (cit. on p. 3).

[PDV15b] Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. “Formal Modelling and Verification of IEC61499 Function Blocks with Abstract State Machines and SMV - Execution Semantics”. In: *Proc. of SETTA’15*. Vol. 9409. LNCS. Springer, 2015, pp. 300–315 (cit. on p. 84).

[PDV15c] Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. “Formal Verification of IEC61499 Function Blocks with Abstract State Machines and SMV - Modelling”. In: *Proc. of TrustCom/BigDataSE/ISPA’15*. IEEE, 2015, pp. 313–320 (cit. on p. 84).

[PVS12] Sandeep Patil, Valeriy Vyatkin, and Majid Sorouri. “Formal verification of Intelligent Mechatronic Systems with decentralized control logic”. In: *Proc. of ETFA’12*. IEEE, 2012, pp. 1–7 (cit. on p. 87).

[PRG20] Eliseu Moura Pereira, João Pedro Correia dos Reis, and Gil Gonçalves. “DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems”. In: *Proc. of SAM-IoT’20*. Vol. 2739. CEUR. CEUR-WS.org, 2020, pp. 63–71 (cit. on p. 63).

[Pin+17] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, et al. “Runtime Enforcement of Cyber-Physical Systems”. In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017), 178:1–178:25 (cit. on p. 90).

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *Proc. of FOCS’77*. IEEE Computer Society, 1977, pp. 46–57 (cit. on pp. 17, 91).

[PHS22a] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. “Real-time Dynamic Reconfiguration for IEC 61499”. In: *Proc. of ICPS’22*. IEEE, 2022, pp. 1–6 (cit. on pp. 4, 88).

[PHS22b] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. “Rollback Sequences for Dynamic Reconfiguration of IEC 61499”. In: *Proc. of INDIN’22*. IEEE, 2022, pp. 81–86 (cit. on pp. 4, 88).

[PS21] Laurin Prenzel and Sebastian Steinhorst. “Automated Dependency Resolution for Dynamic Reconfiguration of IEC 61499”. In: *Proc. of ETFA’21*. IEEE, 2021, pp. 1–8 (cit. on pp. 4, 63, 88).

[Pto14] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014 (cit. on p. 85).

[RH95] M. Rausch and H.-M. Hanisch. “Net condition/event systems with multiple condition outputs”. In: *Proc. of ETFA’95*. Vol. 1. 1995, 592–600 vol.1 (cit. on p. 83).

[RM23] Kaushik Ray and Soumen Moulik. “i-DSME: An industrial-DSME MAC protocol for smart factory automation”. In: *Internet Things* 23 (2023), p. 100859 (cit. on p. 1).

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*. Vol. 4. EATCS Monographs on Theoretical Computer Science. Springer, 1985 (cit. on p. 83).

[RRF20] Matthieu Renard, Antoine Rollet, and Yliès Falcone. “Runtime enforcement of timed properties using games”. In: *Formal Aspects Comput.* 32.2-3 (2020), pp. 315–360 (cit. on p. 60).

[Ros10] William Rosen. *The Most Powerful Idea in the World: A Story of Steam, Industry, and Invention*. Random House, 2010 (cit. on p. 1).

[SAM23] Ashkan Samadi, Marwan Ammar, and Otmane Ait Mohamed. “Statistical Model Checking based Analysis of Fault Trees and Power Consumption to Enhance Autonomous Systems Reliability”. In: *Proc. of NEWCAS’23*. IEEE, 2023, pp. 1–5 (cit. on p. 86).

[SZ11] Andreas Schimmel and Alois Zoitl. “Distributed online change for IEC 61499”. In: *Proc. of ETFA’11*. IEEE, 2011, pp. 1–7 (cit. on p. 88).

[Sch00] Fred B. Schneider. “Enforceable Security Policies”. In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 30–50 (cit. on pp. 48, 91).

[Sch17] Klaus Schwab. *The Fourth Industrial Revolution*. USA: Crown Publishing Group, 2017 (cit. on p. 1).

[Sha+24] Alireza Shahrabi Farahani, Hamed Kohandel, Hamid Moradtabrizi, et al. “Power generation gas turbine performance enhancement in hot ambient temperature conditions through axial compressor design optimization”. In: *Applied Thermal Engineering* 236 (2024), p. 121733 (cit. on p. 3).

[SV21] Viktor Shatrov and Valeriy Vyatkin. “Promela Formal Modelling and Verification of IEC 61499 Systems with comparison to SMV”. In: *Proc. of INDIN’21*. IEEE, 2021, pp. 1–6 (cit. on p. 84).

[SJC14] Roopak Sinha, Kenneth Johnson, and Radu Calinescu. “A scalable approach for re-configuring evolving industrial control systems”. In: *Proc. of ETFA’14*. IEEE, 2014, pp. 1–8 (cit. on p. 89).

[Sin+19] Roopak Sinha, Sandeep Patil, Luís Gomes, and Valeriy Vyatkin. “A Survey of Static Formal Methods for Building Dependable Industrial Automation Systems”. In: *IEEE Trans. Ind. Informatics* 15.7 (2019), pp. 3772–3783 (cit. on pp. 2, 14).

[Sis+18] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. “Industrial Internet of Things: Challenges, Opportunities, and Directions”. In: *IEEE Trans. Ind. Informatics* 14.11 (2018), pp. 4724–4734 (cit. on p. 1).

[Son+21] Lisa Sonnleithner, Bianca Wiesmayr, Virendra Ashiwal, and Alois Zoitl. “IEC 61499 Distributed Design Patterns”. In: *Proc. of ETFA’21*. IEEE, 2021, pp. 1–8 (cit. on pp. 4, 12, 89).

[Son+22] Lisa Sonnleithner, Bianca Wiesmayr, Virendra Ashiwal, et al. “Architectural Concepts for IEC 61499-based Machine Controls: Beyond Normal Operation Handling”. In: *Proc. of ETFA’22*. 2022, pp. 1–8 (cit. on pp. 4, 89).

[SG04] Marius Stanica and Hervé Guéguen. “Using timed automata for the verification of iec 61499 applications”. In: 37.18 (2004). Proc. of IFAC’04, pp. 375–380 (cit. on p. 88).

[Str+08] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, et al. “Framework for Distributed Industrial Automation and Control (4DIAC)”. In: *Proc. of INDIN’08*. 2008, pp. 283–288 (cit. on pp. 10, 62, 64).

[SFV06] Christoph Sünder, Bernard Favre-Bulle, and Valeriy Vyatkin. “Towards an Approach for the Verification of Downtimeless System Evolution”. In: *Proc. of ETFA’06*. IEEE, 2006, pp. 1133–1136 (cit. on p. 88).

[SVZ13] Christoph Sünder, Valeriy Vyatkin, and Alois Zoitl. “Formal Verification of Downtimeless System Evolution in Embedded Automation Controllers”. In: *ACM Trans. Embed. Comput. Syst.* 12.1 (2013), 17:1–17:17 (cit. on pp. 2, 4, 88).

[Tra+20] Duc Do Tran, Jörg Walter, Kim Grüttner, and Frank Oppenheimer. “Towards Time-Sensitive Behavioral Contract Monitors for IEC 61499 Function Blocks”. In: *Proc. of ICPS’20*. IEEE, 2020, pp. 27–34 (cit. on p. 87).

[VH99] V. Vyatkin and H.-M. Hanisch. “A modeling approach for verification of IEC1499 function blocks using net condition/event systems”. In: *Proc. of ETFA’99*. Vol. 1. 1999, 261–270 vol.1 (cit. on pp. 83, 87).

[Vya11] Valeriy Vyatkin. “IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review”. In: *IEEE Trans. Ind. Informatics* 7.4 (2011), pp. 768–781 (cit. on pp. 1, 2).

[WSZ20] Bianca Wiesmayr, Lisa Sonnleithner, and Alois Zoitl. “Structuring Distributed Control Applications for Adaptability”. In: *Proc. of ICPS’20*. Vol. 1. 2020, pp. 236–241 (cit. on pp. 4, 89).

[XPV21] Midhun Xavier, Sandeep Patil, and Valeriy Vyatkin. “Cyber-physical automation systems modelling with IEC 61499 for their formal verification”. In: *Proc. of INDIN’21*. IEEE, 2021, pp. 1–6 (cit. on pp. 30, 86).

[YR10] Li Hsien Yoong and Partha S. Roop. “Verifying IEC 61499 Function Blocks Using Esterel”. In: *IEEE Embedded Systems Letters* 2.1 (2010), pp. 1–4 (cit. on p. 87).

[ZV12] Gulnara Zhabelova and Valeriy Vyatkin. “Multiagent Smart Grid Automation Architecture Based on IEC 61850/61499 Intelligent Logical Nodes”. In: *IEEE Trans. Ind. Electron.* 59.5 (2012), pp. 2351–2362 (cit. on p. 1).

[ZL14] Alois Zoitl and Robert Lewis. *Modelling control systems using IEC 61499. 2nd Edition*. Institution of Engineering and Technology, 2014 (cit. on pp. 10, 46, 47, 64, 67).

[ZP13] Alois Zoitl and Herbert Prähofer. “Guidelines and Patterns for Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language”. In: *IEEE Trans. Ind. Informatics* 9.4 (2013), pp. 2387–2396 (cit. on p. 47).

[ZSE13] Alois Zoitl, Thomas I. Strasser, and Gerhard Ebenhofer. “Developing modular reusable IEC 61499 control applications with 4DIAC”. In: *Proc. of INDIN’13*. IEEE, 2013, pp. 358–363 (cit. on p. 64).

[Zyu+21] Vladimir Zyubin, Igor S. Anureev, Natalya Olegovna Garanina, et al. “Event-Driven Temporal Logic Pattern for Control Software Requirements Specification”. In: *Proc. of FSEN’21*. Vol. 12818. LNCS. Springer, 2021, pp. 92–107 (cit. on p. 91).