Programmable Logic Controllers Past Linear Temporal Logic for Monitoring Applications in Industrial Control Systems

Xia Mao, Xin Li, Yanhong Huang, Jianqi Shi, and Yueling Zhang

***Abstract*—Programmable logic controllers (PLC), which are widely applied in modern industrial control systems (ICS), work as the controller of sensors and actuators in ICS. These systems require strict correctness, especially for safety-critical systems. Currently, increasingly ICS move to "come online" scenarios to enhance cyber-physical features, but it makes them more vulnerable due to acquiring increased interconnection accompanied by weakening physical isolation. Moreover, with the more complex controlling environment, such as hundreds of more I/O points and more diverse field buses, the incorrect executions of PLC might cause the failure of the overall ICS. In this article, we examine how the security and safety of running PLC could be enhanced in both developing and deploying stages of ICS. We propose a novel application of runtime verification to guarantee the security and safety of real-world ICS. As a variant of temporal logic, PLC past linear temporal logic (PPLTL) is proposed to specify the security and safety properties of PLC. Using PPLTL, we synthesize monitors to improve the PLC program's security and safety as a partner of testing and static verification. Our monitors provide twofold processing in a nonintrusive manner: One is filtering abnormal input data before invading the original programs, the other is double-checking the output signals before driving the actuators. We use several case studies and benchmarks to demonstrate the efficiency of the approach. The empirical results show that the time overhead and memory occupation are tiny.**

***Index Terms*—Industrial control system (ICS), international electrotechnical commission (IEC) 61131-3 standard, programmable logic controller (PLC), runtime verification (RV), temporal logic.**

Manuscript received April 9, 2021; revised June 21, 2021 and July 30, 2021; accepted October 19, 2021. Date of publication October 27, 2021; date of current version April 13, 2022. Paper no. TII-21-1608. *(Corresponding author: Jianqi Shi.)*

The authors are with the Software Engineering Institute, East China Normal University, Shanghai 20000, China, and also with the National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, Shanghai 20000, China (e-mail: xia.mao@ntesec.ecnu.edu.cn; xin.li@ntesec.ecnu.edu.cn; yhhuang@sei.ecnu.edu.cn; jqshi@sei.ecnu.edu.cn; ylzhang@smu.edu.sg).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TII.2021.3123194.

Digital Object Identifier 10.1109/TII.2021.3123194

I. Introduction

MANY safety-critical infrastructures and emerging digital factories are under the equipment control of programmable logic controllers (PLC), which act as the on-site "brain" of industrial control systems (ICS). The correctness of their behaviors is highly required since minor faults may cause substantial physical damage, especially in safety-critical systems [3]. Industry 4.0 increasingly introduces cyber-security issues, but few secure mechanisms are considered in the early ICS (e.g., [5]–[8]), thus these legacy ICS show inherent vulnerability [4]. For instance, the traditional SCADA system of distributed control is vulnerable to cyber-physical attacks [4], [6], [9]. In practice, the most commonly used approaches to ensure safety, such as debugging, testing, and simulation, are usually not complete enough to expose potential errors.

Formal methods based on rigorous mathematics, such as model checking and theorem proving [10], are the suggested alternative to guarantee ICS safety. They can formally reveal inconsistencies, ambiguities, and incompleteness that might be difficult for the traditional testing routines. Essentially, *runtime verification* (RV) is a lightweight method branching from modeling checking [16].

Currently, most formal methods applied to PLCs focus on static verification, which seems helpful before deploying systems [11], [12]. For instance, PLCVerif is a model checker that supports a high degree of automation to verify PLC programs, but it largely depends on the program's source code [13]. Moreover, the infamous problem, the so-called state explosion, should be taken care of in model checking since its solving the *language containment* issue is PSPACE-complete reflecting in nondeterministic finite automata [18]. Coq as a prover using strictly mathematical theorem can also verify PLC programs, but these kinds of approaches usually depend on predictable behaviors' description [14]. The above dependence often affects static verification in real-world systems. For instance, it is common to call third-party libraries that do not provide readable source code and behaviors' details. In addition, these static verification approaches do not mean prior guarantee correctness [10].

Typically, RV as a monitoring technique is considered the partner of testing, model checking, and theorem proving, which reaches a beneficial balance between model checking of high complexity and testing with some pitfalls. It deals with the *word acceptance* problem [19] and, therefore, has a lower complexity

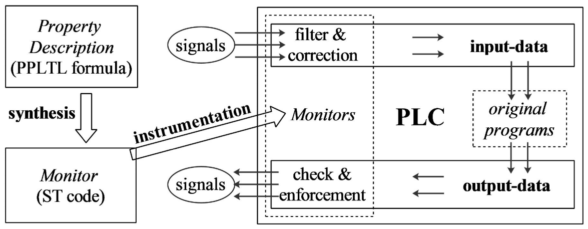


Fig. 1. PLC runtime verification main phases.

(NLOGSPACE-complete) than model checking. The monitoring and the *ad hoc* testing use a similar "black box" technique. The difference exists that the testing requires active input of test cases, but the monitoring passively accepts the generated data during the actual execution. In a sense, the monitoring can be understood as "forever" testing, and therefore it is complete [16].

However, many RV approaches constructed on the general program basis may not meet the needs of the specific PLC-field due to the target-specific language standard, the different execution mechanism from traditional personal computers (PC), the limited resource situation based on embedded devices, and the periodic interaction with peripheral signals. As we know that the particularity of embedded target system in language extensions, hardware combinations, and assembly code would deplete the prospect of the universal approaches [30]. In fact, although RV is always active in communities, the related researches on the PLC field are not abundant. Moreover, most of them use the solution of dedicated external hardware as monitors [5], [8], [31], [32], which might introduce compatibility issues or potential influences on the target system intuitively, such as the electrical characteristics adaptation and the synchronization control for real-time requirements.

Therefore, our work has twofold motivations. One is enhancing PLC execution's security and safety, which is hard to analyze before deployments in static verification, such as anomaly detection and runtime enforcement. The other one is, following the classic processing of generating monitors from logical formulas, such as linear temporal logic (LTL) [20] and past-time LTL (ptLTL) [22], we propose a variant of temporal logic considering the tradeoff between usability and expressiveness in the specific area of PLC. Here, we examine the online monitoring that is attractive to early detection and urgent enforcement rather than offline verification [15].

In this article, we propose a dedicated PLC past linear temporal logic (PPLTL) to describe properties. Then, we present the synthesis of the monitor from the PPLTL formula and the instrumentation mechanism to enhance the security and safety of PLC executions, as depicted in Fig. 1. From the *input* source's perspective, the monitor would filter or correct the potentially insecure data before being used by the original PLC program, such as unstable signals from the failure sensors and the semantic attacks [7] consisting of a series of seemingly valid commands. From the point of the *output* goal, the monitor should act as a final checker just before the output signals are updated to the actuators. When a signal calculated by the PLC is found to violate the property, this is the critical moment to avoid the machine being driven maliciously by immediately alarming and forcing the value to be within the safe range. Moreover, our monitor works in a nonintrusive manner. Namely, the monitoring would neither destroy (e.g., the instrumentation in the middle of code) the integrity of the source program nor depend on any external hardware.

PPLTL extends ptLTL into the PLC execution. The advantages of PPLTL include the following.

1) It extends the proposition of ptLTL to the quantifier-free predicates and maintains the decidability of ptLTL. Compared with some variants that introduce complete first-order predicate logic (e.g., LTLFO [23]), they are undecidable and complicated to use, while our proposed PPLTL examines the balance of expressiveness and practicality. 2) According to the PLC characteristics of the cyclic scanning and sequential execution, the cycle is regarded as an independent clock unit, and PPLTL thereby increases the ability to express time-related properties. 3) Additional counting operators can help satisfy the requirements of statistic and frequency, summarized from the engineering experience of PLC control scenes, such as the workpiece counting and the qualification rate calculating. 4) Some monitoring operators are derived from existing operators to specify properties concisely and quickly. For instance, the rising edge signal $\uparrow G$ written as the standard operators is $\odot(\neg G) \wedge G$, where $\odot$ represents the state of the previous cycle.

Our key contributions are as follows.

1) We propose a novel PPLTL as a formal specification language to describe the security and safety properties of PLC controlling.
2) We propose a framework to synthesize a monitor from the PPLTL formula based on RV.
3) We present an instrumentation mechanism to enhance the PLC execution's security and safety in a nonintrusive (i.e., maintaining the integrity of the original programs) fashion.
4) We demonstrate how to implement the monitoring by several case studies and discuss the quantified overhead among benchmarks.

The next section introduces PLC features, ptLTL basis, and a running example through this article. Section III illustrates the syntax and semantics of PPLTL and presents several applied examples. Section IV demonstrates the synthesis of monitors from PPLTL properties with a proposed framework. In Section V, we present real-life case studies and discuss the results of our benchmarks. Finally, Section VI provides related work, and Section VII concludes this article.

II. Preliminaries

This section introduces the necessary information on PLC and ptLTL. On the one hand, we introduce PLC runtime characteristics, including cyclic scanning and I/O mapping. The I/O mapping would be used for variable binding in the monitoring approach. Moreover, a subsystem of piped gas regulation is used as the running example, and a code segment is written in the structured text (ST) language. On the other hand, we explain why ptLTL is chosen as the basis of extension. The syntax and

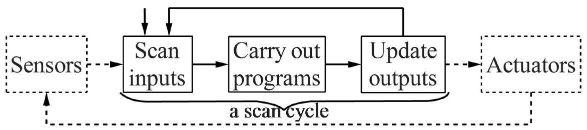
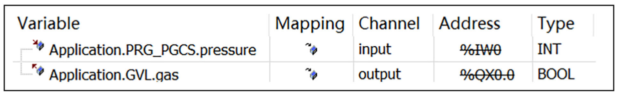


Fig. 2. Scan cycle with three sequential phases.



Variable	Mapping	Channel	Address	Type
Application.PRG_PGCS.pressure		input	~~%IW0~~	INT
Application.GVL.gas		output	~~%QX0.0~~	BOOL

Fig. 3. I/O mapping with an integer variable and a boolean variable.

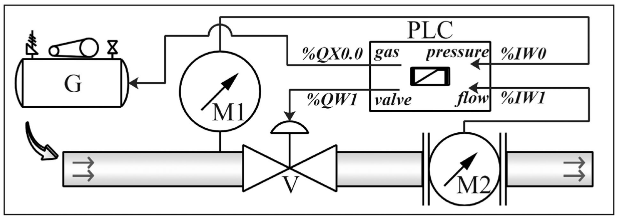


Fig. 4. Example of the piped gas control subsystem (PGCS).

informal semantics are also presented. Finally, several examples show the necessity of temporal logic extension.

A. Programmable Logic Controller

The runtime mechanism of PLC is cyclic scanning, centralized I/O processing, and sequential instructions executing. Fig. 2 shows a scan cycle divided into three steps: *Scan inputs*, *carry out programs*, and *update outputs*. Namely, input and output signals are independently updated before and after programs' calculation within a cycle, respectively. The cyclic value can be fixed by a configuration to represent the cycle's duration. For instance, the well-known platform CODESYS, which supports PLC program development, sets the value to 20 ms in the form of "t#20 ms" by default [1].

The direct addresses of PLC I/O signals are relatively ugly and error-prone. For instance, a Boolean output may look like "%QX0.0." Hence, a more elegant form named I/O mapping sets the variable with a meaningful name to the address. As shown in Fig. 3, a local integer variable named pressure maps to the input "%IW0." A Boolean gas is defined in the global variable list (GVL) corresponding to the output "%QX0.0." This mapping relation is stored in a single file and can also be found in the PLC XML configuration according to IEC 61131-10 [2]. Therefore, I/O mapping would help establish a variable binding between PPLTL formulas and PLC programs. Notably, we only need variable-related configuration but do not care about the specific implementation of programs, and it can be thereby regarded as the "black box" in a sense.

Fig. 4 shows *a running example of the piped gas control subsystem (PGCS)* that regulates gas in pipelines to conform with a specific flow rate and a safe pressure range. G represents the transmission source of gas, $M1$ measures the pressure in the pipelines, and V is a proportional valve to control the gas flow rate measured by $M2$. After the peripherals connected to the PLC configured I/O mapping, the gas variable is BOOL type, and "gas := TRUE" means to open G. The other three variables are INT type, and all correspond to analog signals, which are configured as from 4000 to 20 000 (0%–100%) to represent the 4–20-mA electrical characteristic. For instance, "flow = 4000" means that there is no flowing gas under $M2$ measurement, and "valve := 20 000" means that V is enabled to be fully open.

Example II.1: We use the compact, efficient, and easy-to-read ST language in IEC 61131-3 [2] to realize a control logic that G should be stopped when the $M1$ value exceeds 18 000 as follows:

```
VAR pressure:INT := 0; END_VAR // VAR code
IF pressure > 18000 THEN
 GVL.gas := FALSE; END_IF // BODY code
```

The PLC program consists of the VAR part and the BODY part. Variables and instances are defined between VAR and VAR_END (*line 1*). The rest code is BODY (*lines 2–3*) to implement the logic control with instructions.

B. Past-Time Linear Temporal Logic

LTL is the most used formal language, but the semantics based on the infinite trace makes it unsuitable for monitoring. Hence, variants usually use a finite prefix or past operators, such as LTLFO [23] and ptLTL [22]. Although the past temporal operators will not enhance the expressiveness of LTL, the formula can be exponentially more succinct and visually better understood compared with the pure-future LTL [22]. For instance, the ptLTL formula $\boxdot\,(M2_{\text{flow}} \to \diamondsuit\, V_{\text{open}})$ means that if $M2$ measured results are within regular flow rate, then V should have been opened at some past time, but using LTL would be written as $\neg((\neg V_{\text{open}})U(M2_{\text{flow}} \wedge (\neg V_{\text{open}})))$. $\boxdot$ means all past states, $\diamondsuit$ means some past states, and $\varphi_1\, U \varphi_2$ means keeping φ_1 until φ_2. In addition, compared with the uncertainty of future operators in LTL, the determinism based on past operators makes ptLTL more attractive for monitoring. More precisely, ptLTL formulas can be verified recursively, corresponding to an incremental fashion [21]. The syntax and informal semantics are shown below.

$$\varphi ::= \text{true} \mid \text{false} \mid p \mid \neg\varphi \mid \varphi_1\ op\ \varphi_2 \mid$$
$$\odot\,\varphi \mid \diamondsuit\,\varphi \mid \boxdot\,\varphi \mid \varphi_1\ S\ \varphi_2$$

p is an atomic proposition, and op is a propositional binary operator $(\wedge, \vee, \to)$. A finite trace composed of abstract states (s_i), and $p(s_i)$ equals true if and only if the atomic proposition p holds in the state s_i. Assuming that it is currently in state–i, $\odot\varphi$ is read "previously φ" representing φ should be true in the state–(i-1). $\varphi_1 S \varphi_2$ is read "φ_1 since φ_2" that informally means φ_1 should be true in the past states since (not including) φ_2 was true, such as the trace of "... $\varphi_2\ \varphi_1\ \varphi_1 \ldots \varphi_1$." Besides, $\diamondsuit\ \varphi$ saying "eventually in the past" is equivalent to true$S\varphi$, and $\boxdot\ \varphi$ saying "always in the past" is equivalent to $\neg\diamondsuit\ (\neg\varphi)$. More details of ptLTL are suggested to read [21], [22].

However, ptLTL lacks predicates, so that most practical properties have to transfer the predicate's results to propositional

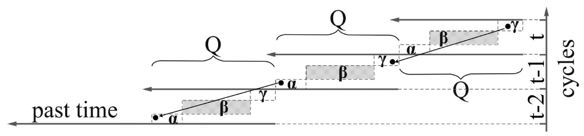


Fig. 5. Trajectory in two-dimensional time-space.

logic tediously. For instance, the too-high pressure cannot be written "$M1 > 18\,000$" directly since the predicate of "$>$" is not supported while only the Boolean result of this expression can be used. In addition, there is not a time or counting operator, and thus, the property related to quantitative time or frequency statistical cannot be expressed directly, which weakens its expressiveness and practicability. For monitoring applications, some derived operators are useful to improve the efficiency of practical use. For instance, $\boxdot\ (M2_{\text{flow}} \rightarrow (\neg V_{\text{close}} \wedge (\odot(\neg V_{\text{close}})SV_{\text{open}})))$ indicates that when $M2$ detects normal flowing gas, V was opened at a certain time and has not been closed until now. It can also be simplified as $\boxdot\ (M2_{\text{flow}} \rightarrow [V_{\text{open}}, V_{\text{close}}))$ by the derived monitoring operator $[\_,\_)$. Informally, $[\varphi_1, \varphi_2)$ means the interval of φ_1 and φ_2 [21]. Therefore, we propose a variant named PPLTL to overcome the above shortcomings.

III. Programmable Logic Controllers Past Linear Temporal Logic

In this work, we propose PPLTL, inspired by the ptLTL application in the monitoring field [21]. Unlike ptLTL, PPLTL supports the predicate, the time operator, and the counter. It also combines PLC features and the formal specification language. Furthermore, we use the cycle's phases, as mentioned in Section II-A, specifying a property for security or safety.

We named these phases α, β, and γ, as shown in Fig. 5. Then, a two-dimensional trajectory consists of the cycles and backward time, which is put in orthogonal dimensions for observation. Specifically, the current cycle is t accumulated from the beginning of PLC execution, and thus, the previous cycle is $t-1$. $\mathcal{Q}$ represents the value of fixed [1] cycle (the default unit is milliseconds), which is the minimum discrete-time granularity of the analysis properties. For instance, we use "$t \times \mathcal{Q}$" to indicate how long the current PLC has been running. Compared with the β phase, where sensors and actuators are isolated, the I/O phases (α and γ) are more natural and economical for monitoring based on PLC characteristics. Note that the I/O phases are handled by the PLC firmware rather than the user program. Thus, a small conversion of the monitoring implementation is introduced in Section IV-C, but it does not affect the use of PPLTL to describe properties.

A. Syntax

As mentioned above, t ($t \in \mathbb{N}$) represents the default time variable, and $\mathcal{Q}$ ($\mathcal{Q} > 0$) represents the default cyclic constant.

[1]For the nonfixed case, $\mathcal{Q}$ could be set to the worst or average cycle duration, but it would introduce the approximation to the monitoring results.

TABLE I
Definition of Derived Monitoring Operators

Formula	Definition	Description
$\uparrow \varphi$	$\neg \odot \varphi \wedge \varphi$	rising edge signals
$\downarrow \varphi$	$\odot\varphi \wedge \neg\varphi$	falling edge signals
$\updownarrow \varphi$	$\odot\varphi \wedge \varphi$	double high signals
$\Updownarrow \varphi$	$\neg \odot \varphi \wedge \neg\varphi$	double low signals
$[\varphi_1, \varphi_2)$	$\neg\varphi_2 \wedge ((\odot\neg\varphi_2)S\varphi_1)$	interval signals
$[\varphi]$	$[\varphi, false)$	enabling signals

Similarly, v is a variable ranging over a finite domain, and c is a constant. In practice, it depends on the specific PLC platform for the accuracy and limits of these variables. For instance, the maximum integer of a 32-bit PLC is $2^{32}-1$. Furthermore, q is a quantifier-free m-ary predicate, and its variables can be regarded as propositional calculus variables. Two predicates are the same if and only if the predicates' names and their variable letters are all the same. For instance, in $q(\kappa_1) \rightarrow q(\kappa_1) \wedge q(\kappa_2)$, $q(\kappa_1)$, and $q(\kappa_2)$ are considered as different propositional variables since κ_1 and κ_2 are not equal. op is a binary operator in $\{\wedge, \vee, \rightarrow\}$. Both $\mathcal{W}$ (read wait) and $\mathcal{Y}$ (read yet) are counters returning counted cycles.

Definition III.1: $\xi_1\mathcal{W}\xi_2$ returns a counter variable $\mathcal{W}_r$ that was initialized to 0. In every cycle, if $\xi_1 = \text{true}$ then $\mathcal{W}_r := \mathcal{W}_r + 1$, next, if $\xi_2 = \text{true}$ then $\mathcal{W}_r := 0$.

Definition III.2: $\xi_1\mathcal{Y}\xi_2$ returns a counter variable $\mathcal{Y}_r$ that was initialized to 0. In every cycle, if $\xi_2 = \text{true}$ then $\mathcal{Y}_r := 0$, next, if $\xi_1 = \text{true}$ then $\mathcal{Y}_r := \mathcal{Y}_r + 1$.

Counting mechanism is that the counter is accumulated by one when ξ_1 is satisfied in the current cycle, but the record is cleared when ξ_2 is satisfied. $\mathcal{W}$ first does ξ_1 and then turns to ξ_2, but $\mathcal{Y}$ is in reverse. Although $\mathcal{W}$ and $\mathcal{Y}$ can represent each other, the different order makes them flexibly meet the needs of whether the current critical cycle is counted (e.g., the triggered alarms) or not (e.g., the time of a pause). Intuitively, when ξ_2 is not satisfied currently, $\mathcal{W} = \mathcal{Y}$ (parameters omitted), otherwise $\mathcal{W} = 0$ and $\mathcal{Y} = 1$. The syntax is defined as follows.

$$\tau ::= t \mid \mathcal{Q} \mid v \mid c$$

$$\xi ::= q(\tau_1, \tau_2, \ldots, \tau_m) \mid \neg\xi \mid \xi_1\ op\ \xi_2$$

$$\kappa ::= \tau \mid \xi_1\mathcal{W}\xi_2 \mid \xi_1\mathcal{Y}\xi_2$$

$$\begin{aligned}\varphi ::= {}& p(\kappa_1, \kappa_2, \ldots, \kappa_n) \mid \neg\varphi \mid \varphi_1\ op\ \varphi_2 \mid \\ & \odot\varphi \mid \Diamond\varphi \mid \boxdot\varphi \mid \varphi_1 S \varphi_2 \mid \\ & \varphi_1, \varphi_2 \mid \uparrow\varphi \mid \downarrow\varphi \mid \updownarrow\varphi \mid \Updownarrow\varphi\end{aligned}$$

$$\phi ::= \alpha.\varphi \mid \gamma.\varphi$$

φ represents the formal property, and the quantifier-free n-ary predicate p takes the values of variables, constants, and counters as its parameters. $\odot$, $\Diamond$, $\boxdot$, and S maintain a similar meaning and naming with ptLTL, and the change is to map an abstract state to a PLC cycle. Hence, they are read as "previous one cycle," "eventually in the past cycles," "always in the past cycles," and " since a cycle," respectively.

The monitoring operations, as shown in Table I, are derived from standard operators. The first four monitor operators are the

TABLE II
SEMANTICS OF PLC PAST LINEAR TEMPORAL LOGIC

$(\epsilon, t) \models p(\kappa_1, \kappa_2, ..., \kappa_n)$	iff	$p(\kappa_1, \kappa_2, ..., \kappa_n) = true$		
$(\epsilon, t) \models \neg\varphi$	iff	$(\epsilon, t) \not\models \varphi$		
$(\epsilon, t) \models \varphi_1 \ op \ \varphi_2$	iff	$(\epsilon, t) \models \varphi_1 \ op \ (\epsilon, t) \models \varphi_2$		
$(\epsilon, t) \models \odot\varphi$	iff	$(\epsilon, t-1) \models \varphi \ for \ t > 1 \ and \ (\epsilon, t-1) = (\epsilon, t) \ for \ t = 1$		
$(\epsilon, t) \models\uparrow \varphi$	iff	$((\epsilon, t-1) \not\models \varphi \ \ (\epsilon, t) \models \varphi) \ for \ t > 1 \ and \ (\epsilon, t-1) = (\epsilon, t) \ for \ t = 1$		
$(\epsilon, t) \models\downarrow \varphi$	iff	$((\epsilon, t-1) \models \varphi \ \ (\epsilon, t) \not\models \varphi) \ for \ t > 1 \ and \ (\epsilon, t-1) = (\epsilon, t) \ for \ t = 1$		
$(\epsilon, t) \models\updownarrow \varphi$	iff	$((\epsilon, t-1) \models \varphi \ \ (\epsilon, t) \models \varphi) \ for \ t > 1 \ and \ (\epsilon, t-1) = (\epsilon, t) \ for \ t = 1$		
$(\epsilon, t) \models\Updownarrow \varphi$	iff	$((\epsilon, t-1) \not\models \varphi \ \ (\epsilon, t) \not\models \varphi) \ for \ t > 1 \ and \ (\epsilon, t-1) = (\epsilon, t) \ for \ t = 1$		
$(\epsilon, t) \models \Diamond \varphi$	iff	$(\epsilon, n) \models \varphi \ for \ some \ 1 \le n \le t$	$iff(recursive)$	$(\epsilon, t) \models \varphi \ or \ ((\epsilon, t-1) \models \Diamond \varphi \ for \ t > 1)$
$(\epsilon, t) \models \boxdot \varphi$	iff	$(\epsilon, n) \models \varphi \ for \ all \ 1 \le n \le t$	$iff(recursive)$	$(\epsilon, t) \models \varphi \ and \ (t > 1 \ implies \ (\epsilon, t-1) \models \boxdot \varphi)$
$(\epsilon, t) \models \varphi_1 S \varphi_2$	iff	$\exists j \le t.((\epsilon, j) \models \varphi_2 \ and$ $\forall k : j < k \le t.(\epsilon, k) \models \varphi_1)$	$iff(recursive)$	$(\epsilon, t) \models \varphi_2 \ or \ ((\epsilon, t) \models \varphi_1 \ and$ $(\epsilon, t-1) \models \varphi_1 S \varphi_2 \ for \ n > 1)$
$(\epsilon, t) \models [\varphi_1, \varphi_2)$	iff	$\exists j \le t.((\epsilon, j) \models \varphi_2 \ and$ $\forall k : j < k \le t.(\epsilon, k) \models \varphi_1)$	$iff(recursive)$	$(\epsilon, t) \not\models \varphi_2 \ and \ ((\epsilon, t) \models \varphi_1 \ or$ $((\epsilon, t-1) \models [\varphi_1, \varphi_2) \ for \ n > 1))$
$(\epsilon, t) \models \alpha.\varphi$	iff	$(\epsilon, t) \models \varphi$		
$(\epsilon, t) \models \gamma.\varphi$	iff	$(\epsilon, t) \models \varphi$		

expression of common signals, such as edge signals. $[\varphi_1, \varphi_2)$ represents the interval situation from the cycle satisfied by φ_1 to the cycle satisfied by φ_2. Specially, we abbreviate $[\varphi, \text{false})$ as $[\varphi]$, which means this property keeps true from the cycle when φ was satisfied. For instance, $[\text{BtnPressed}] \rightarrow \text{SysRunning}$ means this property is violated when a button was pressed, but the system is not running.

The I/O symbols are attached to a property. For a final property formula φ, an identifier of α or γ is added to point out this property should hold for inputting or outputting. For instance, $\alpha.(\text{pressure} < 18\,000)$ means the sampled value (in the input phase) from $M1$ should be less than 18 000.

B. Semantics for PPLTL

We regard a trace as a finite sequence of a state with the corresponding cycle number. These states are generated by the value collected in the α phase and the value output in the γ phase (calculated by the β phase) in a certain cycle. We let ϵ denote the trace $(s_1, 1)(s_2, 2)(s_3, 3)\ldots(s_t, t)$, where t is the current cycle number and s_t is the corresponding state. Then, *the trace ϵ satisfies a formula ϕ in current cycle t is denoted as $(\epsilon, t) \models \phi$.* Table II shows the inductive definition of the semantics, and it also presents the recursive form [21] as the basis for incremental verification.

In the trace ϵ, it satisfies $p(\kappa_1, \kappa_2, \ldots, \kappa_n)$ in cycle t if and only if $p(\kappa_1, \kappa_2, \ldots, \kappa_n)$ is true. Similarly, the satisfaction relation of $\neg$ and *op* can also be obtained. The trace satisfies $\odot\varphi$ if and only if ϵ in cycle $t-1$ satisfies φ, which is similar to the next four $(\uparrow \mid \downarrow \mid \updownarrow \mid \Updownarrow)$. Note that the valuation of the previous cycle for the first cycle equals the first one itself. Namely, $(\epsilon, 1) \models \odot\varphi$ if and only if $(\epsilon, 1) \models \varphi$.

According to the semantics as shown in Table II, a trace $\epsilon = (s_1, 1)(s_2, 2)(s_3, 3)\ldots(s_t, t)$ satisfies the formula $[\varphi_1, \varphi_2)$ if and only if φ_1 was true at some cycle and since then φ_2 has always been false, including in that cycle. Thus, when the trace has only one state (i.e., $t = 1$), it satisfies $[\varphi_1, \varphi_2)$ if and only if $(\epsilon, 1) \models \varphi_1$ and $(\epsilon, 1) \not\models \varphi_2$. When the states of a trace are more than one (i.e., $t > 1$), then first $(\epsilon, t) \not\models \varphi_2$, and then either $(\epsilon, t) \models \varphi_1$, or else the prefix trace satisfies the interval formula, that is, $(\epsilon, t-1) \models [\varphi_1, \varphi_2)$. Other recurrences $(\Diamond \mid \boxdot \mid S)$ can be reasoned with similar processing. The α and γ identifiers will not affect the satisfaction of φ. *Therefore, the computing of whether a trace satisfies the formula only needs states of the current cycle (i.e., cycle–t) and previously adjacent cycle [i.e., cycle–(t-1)].*

C. Applications for PPLTL Formulas

We discuss more applications based on the running example (***Example*** *2.1*). The $\mathcal{Q}$ equals 100 when we acquire the cyclic value that is set to be 100 ms in the XML configuration. Then, there are several application examples as below to specify the properties by PPLTL formulas.

Example III.1: It is a safety property with the predicate and so placed in the output phase. *When the measured $M1$ value is greater than 6000, the closed gas source is temporarily not allowed to open.* That is, the rising edge signal $\uparrow G$ should not appear when $M1 > 6000$, and it is denoted as $\gamma.((M1 > 6000) \rightarrow \neg(\uparrow G))$. If $\uparrow G$ appears due to some defects of the original program, the monitor (introduced in the next section) could enforce the corresponding variable gas to be false for preventing the unsafe opening of the gas source.

Example III.2: It is a frequency-related safety property. According to requirements, it is acceptable that *the occurrence of shutting down the gas source is less than two times per hour.* Namely, from the system beginning, the closing signals $\downarrow G$ of the gas source are monitored. An hour is represented by $36\,000\mathcal{Q}$. It is detonated as $\gamma.(((\downarrow G)\mathcal{Y}(t \mod 36\,000 = 0)) \le 2)$. When the property is violated, the monitor could give an alarm or perform other emergency operations.

Example III.3: It is a time-related safety property. *We stipulate that when the $M2$ value exceeds 11 000, the flow rate is dangerously high so that it should not last more than 10 s (equivalent to $100\mathcal{Q}$).* The property is written as $\gamma.(((M2 > 11\,000)\mathcal{W}(M2 \le 11\,000 \vee \neg G)) \le 100)$. The original program is likely to assume that the valve is always healthy, and the rate can be adjusted to less than 11 000 quickly (within 6 s) and remain relatively stable, as shown in *label 1* of Fig. 6 . However, the potential issue might be ignored. For instance, the adjustment failure is caused by the out-of-control valve, as shown in *label 2*, and so we add this property.

Example III.4: It is a security property based on ***Example*** *3.3*. The property in ***Example****3.3* may be invalid in attacking

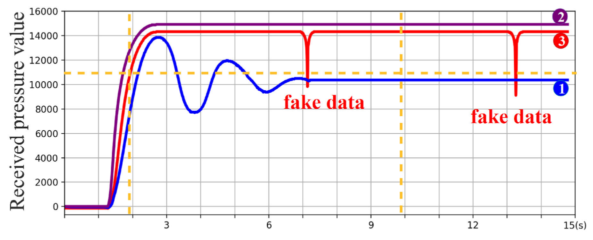


Fig. 6. Pressure value change model under PGCS control. Label 1 indicates normal pressure changes, label 2 indicates the failure of pressure adjustment, and label 3 indicates the failure of pressure adjustment and mixed fake data.

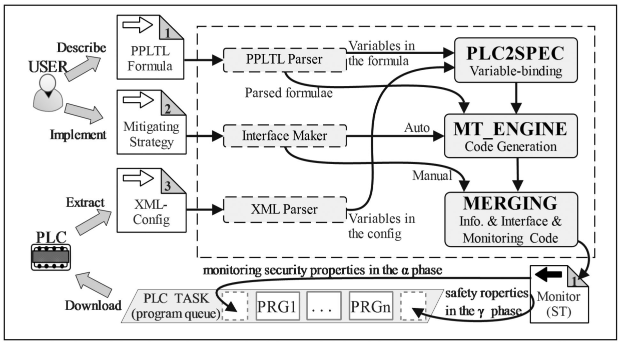


Fig. 7. Framework shows the modules of the monitoring method.

conditions. For instance, in the maliciously infected network, the $M2$ feedback of continuous high flow rate could be mixed with several normal but fake data, as shown in *label 3* of Fig. 6. The fake data will cause the counter to be cleared. To resist the malicious data, we define a more effective property. *Under the gas source opening, if the violated flow rate lasts more than 8 s (adjustable threshold) within 10 s on average, it is then considered a problem.* The corresponding formula is $\alpha.(G \rightarrow (M2 > 11\,000 \mathcal{W}(\uparrow G \vee t \mod 100 = 0)) < 80)$.

IV. Monitoring

A general monitoring approach includes three main parts: The formal formulas specifying concerned properties, the monitors by the synthesis from properties, and the combination mechanisms of monitors and original programs [15], [16], [24]. Fig. 7 shows our framework for the domain-specific monitoring of PLC programs. The properties (*input file1*) are described by PPLTL formulas, and the strategies (*input file2*), such as filtering and enforcement, are used to process the corresponding violated property. According to the IEC 61131-10, we can acquire the program variable, including name, type, and initial value, by the PLC XML-config (*input file 3*). The generated monitors (*output file 1*) depend on the synthesis algorithms with the above three input files. The instrumentation mechanism is nonintrusive that ensures the original program's integrity, where the monitors are added before and after its execution (marked *PLC TASK*) that corresponds to the processing of input and output phases. In the implementation, the monitor insertion is determined by the α or γ identifiers in the head part of the PPLTL formula.

The specific process to generate the monitor is presented in the big dashed box and summarized in three steps. First, the PPLTL formula variables and the program variables are interactively bound by PLC2SPEC, that is, to manually specify the data sources for the property; second, the binding information, the strategy for property violation, and a set of PPLTL formulas rewritten by Algorithm 1 are input to MT_ENGINE, then it generates the monitoring code in ST language. The VAR code is generated by Algorithm 2, and the BODY code is generated by Algorithm 3. Finally, MERGING merges VAR and BODY into a file. Here, we can also add comments and manually supplement more strategies for the violation.

A. Variables Binding and Formulas Rewriting

Extracting variables from the XML-config is a convenient way through a prototype tool automatically, but it can also be completed by manually inputting these variables when there is not a configuration. We define the 3-tuple $(\Theta, \Upsilon, \Gamma)$ to record the necessary information of a variable, where Θ is the variable's name, Υ is the (global or local) position of the configuring, and Γ is the variable's type. For instance, $(\text{gas}, GVL/\text{global}, \text{BOOL})$ indicates that the gas variable with BOOL type is configured in the global file of GVL.

Definition IV.1: When the formula variable Δ is bound to the program variable Θ, the information will be expanded into a 4-tuple $(\Theta, \Upsilon, \Gamma, \Delta)$ and denoted as the bound-config $\mathbb{M}$. Then, $\mathbb{M}(\Theta)\langle\Upsilon\rangle$ and $\mathbb{M}(\Theta)\langle\Gamma\rangle$ represent the position Υ and the type Γ of the variable Θ, respectively. $\mathbb{M}(\Theta)\langle\Delta\rangle$ is the formula variable Δ that is bound with the variable Θ.

Example IV.1: Corresponding to the formula of ***Example**3.4*, the bound-config is $\mathbb{M}_{\text{exp3}} = \{(G, GVL/\text{global}, BOOL, \text{gas}), (M2, \text{PRG\_PGCS/local}, \text{INT}, \text{flow})\}$. Furthermore, $\mathbb{M}_{\text{exp3}}(\text{gas})\langle\Upsilon\rangle$ means a global GVL file, $\mathbb{M}_{\text{exp3}}(\text{flow})\langle\Gamma\rangle$ means INT type, and $\mathbb{M}_{\text{exp3}}(flow)\langle\Delta\rangle$ means $M2$ variable.

Since the monitoring operators are often used in practice, and unifying the formula's temporal operators makes the parsing more convenient, we eliminate some operators by rules according to the provided semantics. Specifically, through rewriting the formula, we only keep (if any) the $\odot$ operator and (five) monitoring operators—$\odot, \uparrow, \downarrow, \updownarrow, \Updownarrow$, and $[\_, \_]$, as shown in the following three rules: 1) $\Diamond\!\!\!\cdot\ \varphi = \varphi \vee \odot[\varphi]$, 2) $\boxdot\ = \varphi \wedge \neg \odot [\neg\varphi]$, and (3) $\varphi_1 S \varphi_2 = \varphi_2 \vee [\odot\varphi_2, \neg\varphi_1)$.

Furthermore, we continue to rewrite the abovepreprocessed formula according to Algorithm 1. The output result is a set of formulas, and each one has a temporal or counting operator at most. *Line 1* is the preprocessing of the formula, and *line 2* defines a subscript variable. *Lines 3–9* split the formula into several subformulas containing a temporal (*Lines 4–5*) or counting (*Lines 6–8*) operator. *Line 10* gets the final formula φ' that does not contain any temporal or counting operator, as well as some subformulas (if any) that are split out.

Example IV.2: Algorithm 1 is applied to parse the formula proposed in ***Example**3.3*, and it acquires the parsed formulas set

Algorithm 1: PPLTL Formula Parsing Algorithm.

```
Input: A PPLTL formula $\varphi_{\text{raw}}$
Output: A set of parsed formulas $\langle \varphi_{s1}, \ldots, \varphi_{sn}, \varphi' \rangle$
 1:  it eliminates $\diamondsuit$ , $\boxdot$ , and $S$ of $\varphi_{\text{raw}}$, and obtains the
     formula $\varphi$
 2:  it initializes the subscript variable $s$ to 1
 3:  while it scans $\varphi$ from left to right and obtains a
     subformula $\varphi_s$ with one temporal or counting
     operatordo
 4:    if ($\varphi_s = k\varphi'_s$ where $k$ is one of $\odot \mid \uparrow \mid \downarrow \mid \updownarrow \mid \Updownarrow$) or
       $\varphi_s = [\varphi'_s, \varphi''_s]$) then
 5:      it renames $\varphi_s$ as $F_s$ and rewrites $\varphi$; $s++$;
 6:    else if $\varphi_s = \varphi'_s\ K\ \varphi''_s$, where $K$ is $\mathcal{W}$ or $\mathcal{Y}$
 7:  it rename $\varphi_s$ as $N_s$ and rewrites $\varphi$; $s++$;
 8:    end if
 9:  end while
10:   the final $\varphi$ is renamed $\varphi'$ that has some (if any)
      subformulas of $F_s$ and (or) $N_s$ (unified denoted as
      $\varphi_{s1}, \varphi_{s2}$,...)
```

$\langle F_1, N_2, \varphi' \rangle$ as follows.

$$\varphi = \alpha.(G \to \underbrace{(M2 > 11\,000)\mathcal{W}(\underbrace{\uparrow G}_{F_1} \vee t \mod 100 = 0))}_{N_2} < 80)$$

$$F_1 = \uparrow G \overset{\text{rewrite}}{\hookrightarrow} \varphi = \alpha.(G \to (M2 > 11\,000)$$
$$\mathcal{W}(F_1 \vee t \mod 100 = 0) < 80)$$

$$N_2 = (M2 > 11\,000)\mathcal{W}(F_1 \vee t \mod 100 = 0) \overset{\text{rewrite}}{\hookrightarrow} \varphi'$$
$$= \alpha.(G \to N_2 < 80).$$

The first scan of φ found that the subformula $\uparrow G$ with one temporal operator (*Line 4*) renamed to F_1 and then updated φ (*Line 5*). The second scan found the counting operator $\mathcal{W}$ (*Line 6*) renamed to N_2 and updated φ (*Line 7*). The third scan did not find any subformula that needs to be renamed and hence jumped out of the loop (*Line 9*). Finally, φ was renamed to φ' (*Line 10*).

B. Synthesis of Monitors

After applying Algorithm 1, we obtain group formulas, such as $\langle F_1, N_2, \varphi' \rangle$, together with the bound-config $\mathbb{M}$, which are taken as the inputs of Algorithm 2. Then, it generates the VAR part of the monitor. We use the form of [command] to indicate that command is the generated code. Specifically, *lines 1* and *21* indicate that the codes between them belong to the VAR part. The data of the previous cycle variables should be stored when some temporal operators appear as shown in *lines 2–6*. "Vv_*pre*" is the historical variable of the previous cycle that corresponds to the program variable v. According to the meaning of each subformula, *lines 7–16* define one or two variables to represent the value of the boolean formula (*Lines 8–12*) or define an integer variable for the counter (*Lines 13–15*). If the default time variable t appears in any formula, *lines 17–19* will define this variable as CYCLE to record the current cycle number. Finally, two internal variables of the monitor are defined in *line 20* to control monitoring switches.

Algorithm 2: Synthesis of VAR Code.

```
Input: A formulas set $\langle \varphi_{s1}, \ldots, \varphi_{sn}, \varphi' \rangle$ and a bound-config
$\mathbb{M}$
Output: The VAR code of the monitor
 1:      [VAR]
 2:        for each variable $\Theta$ in $\mathbb{M}$ do
 3:      if $\mathbb{M}(\Theta)\langle\Delta\rangle$ occurs in $\langle \varphi_{s1}, \ldots, \varphi_{sn} \rangle$ and it is
         restricted by one of $\odot \mid \uparrow \mid \downarrow \mid \updownarrow \mid \Updownarrow$ then
 4:        [$Vv$_pre : $\mathbb{M}(\Theta)\langle\Gamma\rangle$; ]
 5:      end if
 6:    end for
 7:    for each formula $\varphi_s$ in $\langle \varphi_{s1}, \ldots, \varphi_{sn} \rangle$ do
 8:      if $\varphi_s$ has $F'_s$ that is restricted by one of
         $\odot \mid \uparrow \mid \downarrow \mid \updownarrow \mid \Updownarrow$ then
 9:        [$Fs'$_pre : BOOL; ]
10:      end if
11:      if $\varphi_s$ is $F_s$ and $F_s$ has one of $\odot \mid \uparrow \mid \downarrow \mid \updownarrow \mid \Updownarrow, [\_, \_)$
         then
12:        [$Fs$ : BOOL; ]
13:      else if $\varphi_s$ is $N_s$ and $N_s$ has one of $\mathcal{W} \mid \mathcal{Y}$
14:        [$Ns$ : UDINT; ]
15:      end if
16:    end for
17:    if $t$ occurs in $\langle \varphi_{s1}, \ldots, \varphi_{sn}, \varphi' \rangle$ then
18:      [CYCLE : UDINT; ]
19:    end if
20:    [MNT_pre : BOOL := FALSE;
       MNT : BOOL := TRUE; ]
21:    [END_VAR]
```

Example IV.3: The synthesis of VAR code from the property presented in ***Example*3.3** is as follows:

```
VAR
Vgas_pre:BOOL; F1:BOOL; N2:UDINT; CY-
CLE:UDINT;
MNT_pre:BOOL := FALSE; MNT:BOOL := TRUE;
END_VAR
```

The program variable gas in $\mathbb{M}_{\text{exp}}$ corresponds to the formula variable G, and there is a formula "$F1 = \uparrow G$," so the historical variable Vgas_pre is generated (*Lines 2–6*). Then, $F1$ (*Lines 11–12*) and $N2$ (*Lines 13–14*) are the variables corresponding to the subformulas of F_1 and N_2, respectively. We have the CYCLE variable due to the occurrence of the time variable t in the formula N_2 (*Lines 17–19*). Finally, it generates the internal variables of the monitor (*Line 20*).

Proposition IV.1: For a PPLTL formula φ in the trace ϵ, the satisfaction relation of $(\epsilon, t) \models \varphi$ can be recursively calculated by the variables in the cycles of current t and its history $(t - 1)$.

The PPLTL semantics assures the correctness of the above proposition. Meanwhile, it is used as the basis of Algorithm 3 to generate the BODY part. *Line1* replaces all formula variables

Algorithm 3: Synthesis of BODY Code.

```
Input: A formulas set $\langle \varphi_{s1}, \ldots, \varphi_{sn}, \varphi' \rangle$, $\mathbb{M}$, and the VAR
  code.
Output: The BODY code of the monitor
 1: it replaces all variables in $\langle \varphi_{s1}, \ldots, \varphi' \rangle$ with "$Fs$ and
    (or) $Ns$" in VAR and "$\mathbb{M}(\Theta)\langle \Upsilon \rangle.\Theta$" in $\mathbb{M}$
 2: $[IF\ NOT\ MNT\ THEN\ RETURN;\ END\_IF]$
 3: $[IF\ NOT\ MNT$_pre $AND\ MNT\ THEN]$
 4: if VAR contains CYCLE then
 5:    [CYCLE := 0; ]
 6: end if
 7: foreach formula$\varphi_s$ in $\langle \varphi_{s1}, \ldots, \varphi_{sn} \rangle$ do
 8:    if $\varphi_s$ is $Ns$ then
 9:       $[Ns := 0;]$
10:    else if $\varphi_s = [\varphi'_s, \varphi''_s)$
11:       $[\varphi_s :=$ FALSE; ]
12:    else if $\varphi_s = \odot \uparrow | \downarrow | \updownarrow | \Updownarrow \varphi'_s$
13:       $[\varphi'_{\text{s\_pre}} := \varphi'_s;]$ //$\varphi'_{s\_pre}$ is the "pre" form of $\varphi'_s$.
14:    end if
15: end for
16: $[END\_IF]$
17: foreach formula $\varphi_s$ in $\langle \varphi_{s1}, \ldots, \varphi_{sn} \rangle$ do
18:    if $[\mathcal{Y}\ is\ similar.]\varphi_s$ is $Ns$ and $Ns = \varphi'_s \mathcal{W} \varphi''_s$ then
19:       $[IF\ \varphi'_s\ THEN\ Ns := Ns + 1;\ END\_IF$
          $IF\ \varphi''_s\ THEN\ Ns := 0;\ END\_IF]$
20:    else if $\varphi_s$ is $Fs$ and $Fs = [\varphi'_s, \varphi''_s)$
21:       $[Fs := (Fs\ OR\ \varphi'_s)\ AND\ NOT\ \varphi''_s;]$
22:    else if $\varphi_s$ is $Fs$ and $Fs = \odot \varphi'_s$
23:       $[Fs := \varphi'_{s\_pre};]$ //$\varphi'_{s\_pre}$ is the "$pre$" form of $\varphi'_s$.
24:    else if $[\updownarrow, \downarrow, \Updownarrow\ are\ similar.]\varphi_s$ is $Fs$ and
       $Fs = \uparrow \varphi'_s$
25:       $[Fs := NOT\ \varphi'_{s\_pre}\ AND\ \varphi'_s;]$
26:    end if
27: end for
28: $[IF\ MNT$_pre $THEN\ MNT := \varphi'$(that ignores $\alpha$
    or $\gamma$);
 $IF\ MNT\ THEN\ (*SAT\_ITF*)\ ELSE\ (*UNSAT\_ITF*)$
 $END\_IF\ END\_IF\ MNT_{\text{\_pre}} := MNT;]$
29: for all $\varphi'_{s\text{\_pre}}$ in the above processing do
30:    $[\varphi'_{s\text{\_pre}} := \varphi'_s;]$
31: end for
32: if VAR contains CYCLE then
33:    [CYCLE := CYCLE + 1; ]
34: end if
```

with the original program variables and the monitor variables defined in VAR. For instance, G should be replaced with GVL.gas, and F_1 should be replaced with $F1$. *Lines 2–16* initialize some variables in the monitor, and this processing happens in the PLC starting and the monitor resetting. $\varphi'_{s\text{\_pre}}$ is the "pre" form of φ'_s to record the previous value of φ'_s, such as Vgas_pre for gas and $F2$_pre for $F2$. *Lines 17–27*, according to semantics, convert the counting (*Lines 18–19*) or temporal (*Lines 20–26*) operators of subformulas into code implementation in an incremental manner. *Line 28* handles interfaces for property satisfaction or violation. It is only executed after the monitor is initialized because of the control of variables MNT_pre and MNT. Usually, for the developing programs, excepting the necessary logging, "MNT := FALSE" can be added to the UNSAT_ITF that stops the monitor as a violation occurs. After the problem was confirmed or cleared, we can restart the monitor by "MNT := TRUE." For deployed ICS, the monitoring can keep running, and it executes the mitigating strategies during cycles of violation. *Lines 29–31* update the data of historical variables in the monitor. The cycle's number is recorded via the variable CYCLE if the monitor needs it (*Lines 35–37*).

Example IV.4: The synthesis of BODY code from the property shown in ***Example**3.4* is listed below. In the listing code, *line 1* decides whether to enable the monitor or not, and *lines 2–6* initialize variables. *Line 7* corresponds to compute the Boolean result of formula F_1, and *lines 8–13* corresponds to calculate the integer result of formula N_2. Here, we have implemented the violation interface in *lines 14–21*, which has a warning command "GVL.alarm := TRUE" supported by the original program and an enforcement command "GVL.gas := FALSE" to close the gas source. Finally, *lines 22–23* update two historical variables.

```
IF NOT MNT THEN RETURN;END_IF
IF NOT MNT_pre AND MNT THEN
 CYCLE := 0;
 Vgas_pre := GVL.gas;
 N2 := 0;
END_IF
F1 := NOT Vgas_pre AND GVL.gas;
IF PRG_PGCS.flow > 11000 THEN
 N2 := N2 + 1;
END_IF
IF F1 OR CYCLE MOD 100 = 0 THEN
 N2 := 0;
END_IF
IF MNT_pre THEN
 MNT := NOT GVL.gas OR N2 < 80;
 IF MNT THEN (*SAT_ITF*)
 ELSE (*UNSAT_ITF*)
 GVL.alarm := TRUE;
 GVL.gas := FALSE;
 END_IF
END_IF
MNT_pre := MNT;
Vgas_pre := GVL.gas;
```

C. Combinations of Monitors and PLC Programs

As mentioned in Section III, the handling of I/O phases could be seen as controlled independently by the PLC firmware, which is not programmable to users. Therefore, Fig. 8 shows the conversion in actual implementation. The monitor marked α should be postponed to be placed at the head of the β phase. Since sampling input signals have not entered the original programs, any insecure data causing the property violation can be timely filtered by monitors' interface implementation. Similarly, the monitoring for γ should be advanced to the tail of the β phase. Thus, the updating of driving signals can be rechecked by

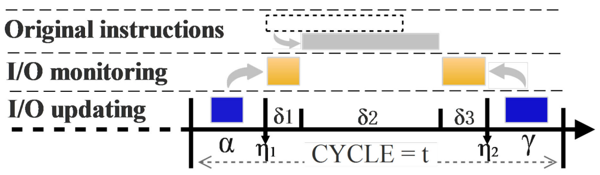
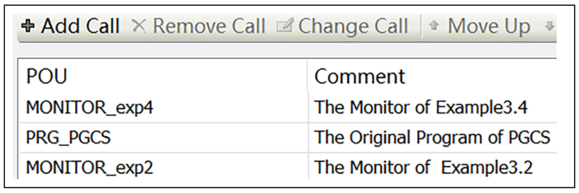
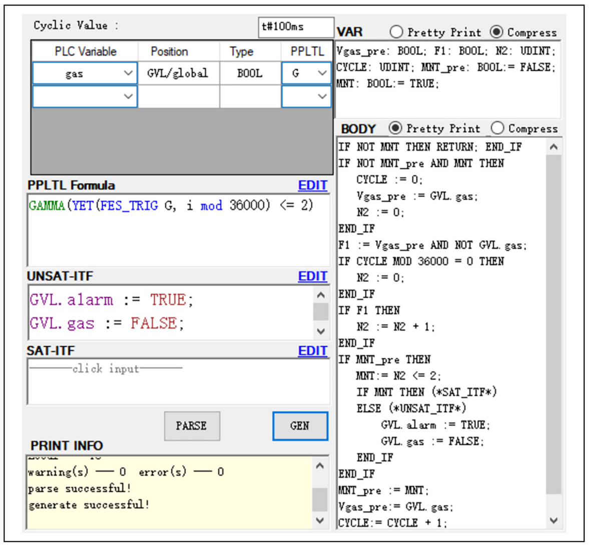


Fig. 8. Implementation of three phases in programming environment.



POU	Comment
MONITOR_exp4	The Monitor of Example3.4
PRG_PGCS	The Original Program of PGCS
MONITOR_exp2	The Monitor of Example3.2

Fig. 9. Placed position of the monitors and the original program.



Cyclic Value : t#100ms

PLC Variable	Position	Type	PPLTL
gas	GVL/global	BOOL	G

PPLTL Formula

```
GAMMA(YET(FES_TRIG G, i mod 36000) <= 2)
```

UNSAT-ITF

```
GVL.alarm := TRUE;
GVL.gas := FALSE;
```

SAT-ITF

```
click input
```

PRINT INFO

```
warning(s) —— 0  error(s) —— 0
parse successful!
generate successful!
```

VAR ○ Pretty Print ● Compress

```
Vgas_pre: BOOL; F1: BOOL; N2: UDINT;
CYCLE: UDINT; MNT_pre: BOOL:= FALSE;
MNT: BOOL:= TRUE;
```

BODY ● Pretty Print ○ Compress

```
IF NOT MNT THEN RETURN; END_IF
IF NOT MNT_pre AND MNT THEN
   CYCLE := 0;
   Vgas_pre := GVL.gas;
   N2 := 0;
END_IF
F1 := Vgas_pre AND NOT GVL.gas;
IF CYCLE MOD 36000 = 0 THEN
   N2 := 0;
END_IF
IF F1 THEN
   N2 := N2 + 1;
END_IF
IF MNT_pre THEN
   MNT:= N2 <= 2;
   IF MNT THEN (*SAT_ITF*)
   ELSE (*UNSAT_ITF*)
      GVL.alarm := TRUE;
      GVL.gas := FALSE;
   END_IF
END_IF
MNT_pre := MNT;
Vgas_pre:= GVL.gas;
CYCLE:= CYCLE + 1;
```

Fig. 10. PLCmnt prototype to generate the monitoring module.

the monitor. The planning outputs are unsafe and violate the property, and then the signals would be enforced to safer ones before the real outputting. For instance, a property is that two coils of a dual-coil solenoid valve are forbidden to be enabled together, and then the monitor could disable one of them when that happens.

Example IV.5: Fig. 9 shows that we have inserted the synthesized monitors, corresponding to ***Example****3.4* and ***Example****3.2*, to the original program's front and back. Thus, during one scan cycle, after the monitor MONITOR_exp4 finishing, the original PRG_PGCS then executes and finally turns to the monitor MONITOR_exp2.

Proposition IV.2: In one cycle, assuming that the β phase time is from η_1 to η_2 with the strict relation $\eta_1 < \eta_2$, and the time overhead of the security and safety monitoring is δ_1 and δ_3, respectively. Then, the PLC mechanism requires that the rest time δ_2 (equals $\eta_2 - \eta_1 - \delta_1 - \delta_3$) should be enough for the execution of the original program. Otherwise, it is unsafe.

A severe issue for monitoring is that implanted monitors might influence the overall system behavior [16]. Thus, ***proposition*** *4.2* is significant in practice because it intuitively informs the affecting condition between the monitoring execution and the original PLC program, which helps us evaluate monitors' feasibility in real-world ICS. Furthermore, our synthesized monitor uses as few resources as possible to weaken the impact of implantation, and the advantages are summarized as the following three points: 1) It is an efficient algorithm using an incremental verification; 2) it directly uses the sharing variables of the original program, and it isolates the impact to these variables through read-only and sequential execution (no data race); 3) it only generates necessary historical variables to save storage. More overhead details are discussed in several experiments in the next section.

V. Application

We show the synthesis applying for PGCS based on ***Example*** *3.2* through our prototype tool PLCmnt, which implements the proposed framework. Another case study of the temperature control system (TCS) is also demonstrated to detect the potential semantic attack. Finally, to illustrate our approach's efficacy, several benchmarks are analyzed, including the above two case studies and some more.

A. Automatic Synthesis of Monitors for PGCS

Fig. 10 shows the automatic synthesis of the monitor by PLCmnt, from the formula $\gamma.((\downarrow G)\mathcal{Y}(t \mod 36\,000) \leq 2)$ in ***Example*** *3.2*. The cyclic value is set to 100 ms, and the formula variable G binds to the program variable gas in the GVL file. We implemented the violation interface that contains triggering alarms (GVL.alarm := TRUE) and enforcing the disabling signal (GVL.gas := FALSE) to prevent the gas from the unsafe transmission. The generated monitor is presented in two parts, namely, VAR and BODY.

B. Attack Detection for Temperature Control System

TCS has a cyclic value of 20 ms. There are five used I/O variables here—rTempSet, bLockMode, rTempSensor, bCompressor, and rHysteresis. Specifically, rTempSet represents the newly received temperature to be set, but it works when bLockMode is equal to true. rTempSensor feedbacks the current environment temperature. rHysteresis provides a rest range for the compressor that is driven by bCompressor. This processing is used to prevent frequent compression. For instance, "rTempSet=200 (Kelvin)" and "rHysteresis=20" means a compressor starts to cool when the ambient temperature exceeds 220 K until it falls to 180 K. Here, TCS sets a 4-s delay to eliminate the adjustment jitter when setting a new temperature. Similar delays are often used in ICS but might become attack objects.

Due to the lack of response plans to (abnormal) long-term jitter in TCS, the new temperature updating may be continuously delayed when a malicious replay temperature signal is received through the infected network. For instance, the Modbus network can be injected malicious data by remote attackers since it has no security mechanisms [9]. As a result, some hosts or devices

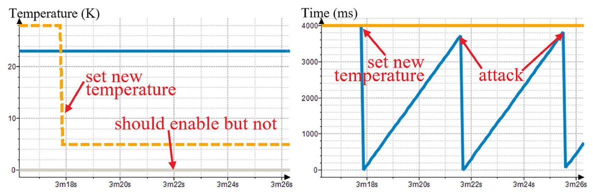
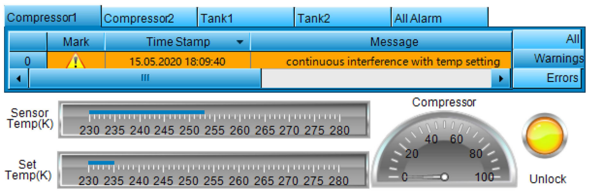


Fig. 11. Diagram of the signal evolution during the attack. The left picture shows the compressor dose not work in time, and the right picture shows the reason is that the timer (delay) is repeatedly reset.



	Mark	Time Stamp	Message
0	⚠	15.05.2020 18:09:40	continuous interference with temp setting

Fig. 12. Display panel of HMI of TCS.

might be infected to be man-in-the-middle (MitM) and launch the baseline response replay attack [17]. Therefore, we specified a safety property to the replay attack, the compressor should work within 10 s when the ambient temperature is higher than the range of a new set temperature. Otherwise, the strategy filters out the set signal so that the new temperature takes effect. The bound-config $\mathbb{M}_{\text{TCS}}$ and the formula φ are as shown below

$$\begin{aligned}\mathbb{M}_{\text{TCS}} = \{&(\text{bLockMode},\ \text{GVL/global},\ \text{BOOL},\ p),\\ &(\text{rTempSet},\ \text{GVL/global},\ \text{REAL},\ a),\\ &(\text{rTempSensor},\ \text{GVL/global},\ \text{REAL},\ c),\\ &(\text{rHysteresis},\ \text{PLC\_PRG/local},\ \text{REAL},\ b),\\ &(\text{bCompressor},\ \text{GVL/global},\ \text{BOOL},\ q)\}\\ \varphi = \alpha.((\neg p \wedge \neg q) &\rightarrow (([a+b \leq c, q)\mathcal{W}q) < 500)).\end{aligned}$$

Fig. 11 shows our simulation of the replay attack. When MitM sniffed the message of temperature setting (to V_t), it modified the value to $V_t \pm 0.01$ and then replayed the message every less than 4 s. It caused the failure of the temperature setting for TCS. However, it was quickly detected by our monitor, and the popped warning can be seen in the human-machine interface (HMI) in Fig. 12.

C. Evaluation and Discussion

Experiments were performed on a machine of Intel(R) i7-7660 U CPU @ 2.50 GHz with 16 GB RAM and the operating system of Windows 10 Multiple Editions. The PLC development system is CODESYS V3.5 SP11 Patch6 (64-bit), which also provides the soft-PLC (64KB+ memory) configured as CODESYS Control RTE V3 x64 (3.5.11.60) [1]. The simulation environment is based on the CODESYS HMI and the simulator of Factory I/O V2.4.3.

TABLE III
BENCHMARKS FOR THE PRESENTED APPROACH

PLC Program		PPLTL Formula			Monitor Overhead			
Name	Cycle (ms)	Timed (?)	Count (?)	Size	GD (bytes)	GC (bytes)	Exec. (µs)	Prop. (%)
PGCS_{exp1}	20	No	No	3	94	508	19	0.095
PGCS_{exp2}	20	Yes	Yes	4	102	620	19	0.095
PGCS_{exp3}	20	No	Yes	4	96	524	22	0.11
PGCS_{exp4}	20	Yes	Yes	5	102	652	41	0.205
TCS	20	No	Yes	9	112	676	39	0.195
OLCS	20	No	Yes	3	115	580	22	0.11
TLS	100	No	Yes	48	141	1492	22	0.022
RS	200	Yes	Yes	7	119	780	22	0.011
FES	50	No	No	7	107	572	18	0.036

Oil Level Control System (OLCS) is a common type of process control system. Here, it is required to control tanks' liquid levels. When the level is too-high due to the liquid injection, the normal oil drain processing can reduce the level below 80% within 10 s. Otherwise, it emits warnings. *Traffic lights system (TLS)* is a typical example frequently used in researches (e.g., in [5]). It controls the four groups of lights at the intersection as the "red-green-yellow" mode, and the lighting colors of vertical and horizontal lights are mutually exclusive. One of the time-related properties is that each light has its maximum duration of individual lighting. Both *refrigerator system (RS)* and *fire engine system (FES)* are classic demonstration projects provided by CODESYS on the platform itself. In *RS*, the cooling process should occur at a lower frequency (every 30 min) when the sensor shows that the door is closed. *FES* controls the ladder car. The signal for extending or retracting the ladder (same as turning left or right) cannot be enabled simultaneously. When the ladder is fully extended, and the boom motion is less than 70 degrees, the turning movement is dangerous.

Both generated data (GD) and generated code (GC) are substantial indicators usually used to evaluate the hardware resource occupancy of the target programs. They can be obtained directly from the compilation log of CODESYS, and other platforms are similar. Here, the "Size" of the formula is intuitively defined as the total occurrences of variables, counters, and temporal operators. Usually, large-size formulas may tend to consume more hardware resources because more variables and instructions are generated in the monitor. As shown in Table III, even for the largest (size is 48) sample TLS, the sum of GD (141 bytes) and GC (1492 bytes) occupies less than 2.5% resources for a general (like 64-KB memory) PLC. *It can be seen that each compiled monitor is quite small.*

The execution time of the monitor (indicated by "Exec." in every cycle) is a critical indicator to most RV approaches because it, to some extent, reflects the impact on the original programs. The columns of "Timed" and "Count" mark whether there are time variables and counting operators in the formula, respectively. According to our algorithms, the formula for the property with timing or counting would introduce new variables, such as F_1 and N_2, for rewriting and thereby may tend to need more execution time. However, from the experimental results, *the time-consuming of the monitor is relatively small on the whole*—the maximum is not more than 41 ms.

The bounds of monitoring are twofold. An intuitive distinguishable bound is whether the hardware resources of the PLC can meet the hardware overhead of the monitor (DC+DG). If the resources are insufficient, COCDESYS will directly display the compilation failure. The other, as pointed out in **Proposition***4.2*, is that the total time of original consumption and additional monitoring cannot exceed one scan cycle. The last column (marked as "Prop.") shows the (very small) percentage of the monitor's maximum execution time in a scan cycle. In practice, provided that we intentionally reduce the scan cycle or add dummy instructions (such as meaningless loops) to the monitor to break **Proposition***4.2*, the corresponding result is that the PLC (by the firmware mechanism) will trigger an incomplete execution alert.

The time-related variable (t) appearing in formulas may tend to increase the consumption of hardware resources and execution time. On the one hand, The time variable in the monitor is of the same type as the count variable, that is, the larger integer UDINT (32 bits). On the other hand, the timing property might use time-consuming arithmetic, such as modulo operation. Corresponding to the experimental results, the overhead of $\mathrm{PGCS}_{\mathrm{exp1}}$ and FES without time variable and counter is relatively small.

To summarize, all monitors successfully detect the violation, and the strategies timely mitigate further damage. Furthermore, the proposed approach is generally scalable since the overhead of monitors is relatively low, and practical applications usually reserve a certain amount of resources and cycle time.

VI. Related Work

As a formal method, RV has been applied in diverse and active fields in the past fifteen years [24], especially for untrustworthy or safety-critical systems. It also alleviates the security and safety issues of ICS [5], and its related work is shown below.

A. Static Verification and Runtime Verification

For the stage before deploying the PLC system, many formal approaches are based on model checking and theorem proving to verify the PLC program statically [11], [12]. PLCVerif is a well-known model checker, which is promising in industries [13]. Blech *et al.* [14] showed how to improve the correctness of PLC programs by the prover of Coq.

In fact, complex systems are often hard to predict or analyze prior to execution, especially for systems with the characteristics of adaptability and self-management [16], but RV is a great option to supplement it. Falcone *et al.* [25] classified and compared the existing tools based on RV in a survey. It also shows that RV-based approaches are popular in many communities, but most of them focus on solving general issues and focus on common programming of high-level languages. That is, RV is rarely used for domain-specific applications (e.g., R2U2 for the unmanned aerial vehicle domain [26]) including the PLC field. More importantly, the combination of RV and the specific field would make the approach more flexible, professional, and practical since the features of this field are considered profoundly.

B. External Monitoring and Local Monitoring

For the monitored system, it is the only simple way to eliminate the introduced affection of the monitor by using external dedicated monitoring hardware [16]. Most of the RV-based approaches for PLC use the external solution [5], [8], [31], [32]. Typically, Jin *et al.* [8] demonstrated an approach of forwarding PLC I/O data to the server that simulates the corresponding controller. Caselli *et al.* [35] extracted helpful events with time-related information from the network traffic traces, and then they constructed the discrete-time Markov Chain (DTMC) model, which describes the normal message sequence and is used to identify semantic attacks. The same based on the PLC traffic, Kleinman and Wool [32] proposed an approach of intrusion detection, which uses the model of deterministic finite automaton to monitor various channels on the S7 PLC Platform. Mercaldo *et al.* [33] presented a study of SCAD attack detection based on modeling with timed automata (executed in UPPAL). In addition, Janicke *et al.* [34] demonstrated a study of deploying monitors on independent Arduino hardware. The monitor is based on a verification framework named Tempura, which can be used to provide early warning for systems such as SCADA.

However, as shown in the survey [27], external monitors are often implemented based on the additional computing hardware and network environment, which would inevitably introduce a synchronous or asynchronous mechanism such as checkpoints. From this point of view, the local monitor can achieve real-time verification more naturally because it saves the cost of external communication. More importantly, the communication may introduce new vulnerable targets. The research of Garcia *et al.* [28] showed that the increasing power of PLC computation makes it possible to perform on-device monitoring. Their approach needs to use embedded hypervisors that can integrate external library functions into PLC execution. In detail, they use the Windows dynamic-link library (DLL) to verify whether the refreshed I/O signals are in a safe range, while the disadvantage is that PLC may break down when some DDLs are abnormal. Our approach of local monitoring does not use any external hardware resources nor depend on additional hypervisors.

C. Temporal Logics and State Machines

Both state machine (i.e., automata) and temporal logic are classic languages in RV, but their modeling focuses are different [15]. Specifically, the former is an executable language whose constructed model can be directly executed as a monitor, but this process is error-prone for complex systems since it lacks abstraction ability in a sense [5], [32], [33]. The latter is a declarative language, which can naturally describe properties according to actual needs, but this approach usually requires the development of monitor synthesis algorithms and development frameworks. As discussed in [15], *temporal logic is generally more convenient than automata* by specifying properties at abstract levels.

A comparative example is that Pearce *et al.* [5] proposed a novel approach for PLC monitor. It is based on the valued discrete timed automata and works in independent hardware, becoming smart I/O. It presented a running example that the system satisfies two properties: 1) The set current i_{set} cannot be greater than the maximum safe current $i_{\max}$; 2) the current i cannot be continuously overloaded for more than time τ. Then, the constructed automaton has *three states, eight transitions, and several guards*, while PPLTL succinctly writes it as $\alpha.(i_{\text{set}} \leq i_{\max} \wedge (i > i_{\text{set}} \mathcal{W} i \leq i_{\text{set}}) \leq \tau)$ using PPLTL. Moreover, [5] uses field programmable gate array as external hardware to avoid occupying PLC resources. In practice, it should solve adaptation issues, such as peripherals' electrical characteristics and PLC communication protocols. Also, it may increase the cost of handling the unknown compatibility problems.

VII. Conclusion

This work proposed PPLTL to specify the properties of PLC programs easily. Then, we proposed a framework for synthesizing monitors from PPLTL formulas, which is implemented by a prototype. Moreover, it shows how to implant the monitor in the system in a nonintrusive manner. Several case studies are demonstrated how this approach improves the security and safety of ICS. The benchmarks are examined that this approach consumed little time and occupied few hardware resources. Thus, it may have great potential to be adopted by ICS.

The future work is twofold. One is to continually strengthen the relevant content of the theory for the proposed logic. We plan to develop an axiomatic system of PPLTL, which will be used to provide a proof system for the conjectured potential properties. Then, we can study the soundness and completeness of PPLTL. The other is to optimize our monitoring algorithm to handle critical resources. We plan to refine the monitor with a tradeoff configuration, which can adjust the balance between time overhead and verification integrity (see [29] or similar). For example, we can set the limit of the maximum runtime of the monitor.

References

[1] CODESYS Group, Kempten, Germany. *Codesys Development System*. [Online]. Available: https://www.codesys.com/

[2] International Electrotechnical Commission, "IEC International Standard IEC 61131," IEC, 2019. [Online]. Available: https://plcopen.org/

[3] D. J. Smith and K. G. Simpson, *The Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety: IEC 61508 (2010 Edition), IEC 61511 (2015 Edition) and Related Guidance*. Oxford, U.K.: Butterworth-Heinemann, 2020.

[4] D. Bhamare, M. Zolanvari, A. Erbad, R. Jain, K. Khan, and N. Meskin, "Cybersecurity for industrial control systems: A survey," *Comput. Secur.*, vol. 89, 2020, Art. no. 101677.

[5] H. Pearce, S. Pinisetty, P. S. Roop, M. M. Kuo, and A. Ukil, "Smart I/O modules for mitigating cyber-physical attacks on industrial control systems," *IEEE Trans. Ind. Informat.*, vol. 16, no. 7, pp. 4659–4669, Jul. 2020.

[6] G. Loukas, *Cyber-Physical Attacks: A Growing Invisible Threat*. Oxford, U.K.: Butterworth-Heinemann, 2015.

[7] Y. Hu, Y. Sun, Y. Wang, and Z. Wang, "An enhanced multi-stage semantic attack against industrial control systems," *IEEE Access*, vol. 7, pp. 156 871–156 882, 2019.

[8] C. Jin, S. Valizadeh, and M. van Dijk, "Snapshotter: Lightweight intrusion detection and prevention system for industrial control systems," in *Proc. IEEE Ind. Cyber- Phys. Syst.*, 2018, pp. 824–829.

[9] Y. Mo *et al.*, "Cyber-physical security of a smart grid infrastructure," *Proc. IEEE*, vol. 100, no. 1, pp. 195–209, Jan. 2012.

[10] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, 1996.

[11] A. Mader, "A classification of PLC models and applications," in *Discrete Event Systems*. Boston, MA, USA: Springer, 2000, pp. 239–246.

[12] R. Sinha, S. Patil, L. Gomes, and V. Vyatkin, "A survey of static formal methods for building dependable industrial automation systems," *IEEE Trans. Ind. Informat.*, vol. 15, no. 7, pp. 3772–3783, Jul. 2019.

[13] D. Darvas, E. Blanco, and S. V. Molnár, "PLCverif re-engineered: An open platform for the formal analysis of PLC programs," in *Proc. 17th Int. Conf. Accel. Large Exp. Phys. Control Syst.*, 2019, pp. 21–27.

[14] J. O. Blech and S. O. Biha, "On formal reasoning on the semantics of PLC using COQ," 2013, *arXiv:1301.3047*.

[15] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification*. Cham, Switzerland: Springer, 2018, pp. 1–33.

[16] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[17] P. Radoglou-Grammatikis, I. Siniosoglou, T. Liatifis, A. Kourouniadis, K. Rompolos, and P. Sarigiannidis, "Implementation and detection of Modbus cyberattacks," in *Proc. 9th Int. Conf. Modern Circuits Syst. Technol.*, 2020, pp. 1–4.

[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA, USA: Addison-Wesley, 1979.

[19] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[20] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annu. Symp. Found. Comput. Sci.*, 1977, pp. 46–57.

[21] K. Havelund and G. Roşu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools Technol. Transfer*, vol. 6, no. 2, pp. 158–173, 2004.

[22] F. Laroussinie, N. Markey, and P. Schnoebelen, "Temporal logic with forgettable past," in *Proc. 17th Annu. IEEE Symp. Log. Comput. Sci.*, 2002, pp. 383–392.

[23] J.-C. Kuester, "Runtime verification on data-carrying traces," Ph.D. dissertation, College Eng. Comput. Sci. Australian Nat. Univ., Canberra, ACT, Australia, 2016.

[24] O. Sokolsky, K. Havelund, and I. Lee, "Introduction to the special section on runtime verification," *Int. J. Softw. Tools Technol. Transfer*, vol. 14, pp. 243–247, 2012.

[25] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools," in *Proc. Int. Conf. Runtime Verification*, 2018, pp. 241–262.

[26] K. Y. Rozier and J. Schumann, "R2U2: Tool overview," in *Proc. RV-CuBES. Int. Workshop Competitions, Usability, Benchmarks, Eval., Standardisation Runtime Verification Tools*, 2017, pp. 138–156.

[27] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "A survey of runtime monitoring instrumentation techniques," 2017, *arXiv:1708.07229*.

[28] L. Garcia, S. Zonouz, D. Wei, and L. P. De Aguiar, "Detecting PLC control corruption via on-device runtime verification," in *Proc. Resilience Week*, 2016, pp. 67–72.

[29] T. Zhang, G. Eakman, I. Lee, and O. Sokolsky, "Overhead-aware deployment of runtime monitors," in *Proc. Int. Conf. Runtime Verification*, 2019, pp. 375–381.

[30] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski, "Past time LTL runtime verification for microcontroller binary code," in *Proc. Int. Workshop Formal Methods Ind. Crit. Syst.*, 2011, pp. 37–51.

[31] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A trusted safety verifier for process controller code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 86–100.

[32] A. Kleinman and A. Wool, "Accurate modeling of the siemens S7 SCADA protocol for intrusion detection and digital forensics," *J. Digit. Forensics, Secur. Law: JDFSL*, vol. 9, no. 2, pp. 37–50, 2014.

[33] F. Mercaldo, F. Martinelli, and A. Santone, "Real-time SCADA attack detection by means of formal methods," in *Proc. IEEE 28th Int. Conf. Enabling Technol.: Infrastructure Collaborative Enterprises*, 2019, pp. 231–236.

[34] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-monitoring for industrial control systems," *Electronics*, vol. 4, no. 4, pp. 995–1017, 2015.

[35] M. Caselli, E. Zambon, and F. Kargl, "Sequence-aware intrusion detection in industrial control systems," in *Proc. 1st ACM Workshop Cyber- Phys. Syst. Secur.*, 2015, pp. 13–24.

Xia Mao received the B.S. degree in software engineering from the Jiangxi University of Finance And Economics, Jiangxi, China, in 2016, and is currently working toward the Ph.D. degree in computer science with East China Normal University, Shanghai, China.

His research interests include formal method, runtime verification, IEC 61508 and IEC 61131 standards, and related applications for safety-critical software systems.

Xin Li received the B.S. degree in software engineering, in 2009, from East China Normal University, Shanghai, China, where he is currently working toward the graduate degree with the School of Computer Science and Software Engineering.

His research interests include formal method and runtime verification.

Yanhong Huang received the B.S. degree in software engineering and the Ph.D. degree in computer science from East China Normal University, Shanghai, China, in 2009 and 2014, respectively.

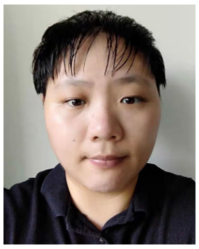
She is an Associate Researcher with Software Engineering Institute, East China Normal University. From February 2012 to July 2012, she was a Research Student with Teesside University, Middlesbrough, U.K. Her research interests include formal method, semantics theory, and analysis and verification of embedded systems and industry software.

Dr. Huang was the Recipient of National Scholarship Award, in 2013, IBM China Excellent Students, in 2013, and Shanghai Excellent Graduates, in 2009 and 2014, respectively.

Jianqi Shi received the B.S. degree in software engineering and the Ph.D degree in computer science from East China Normal University, Shanghai, China, in 2007 and 2012, respectively.

He is currently an Associate Researcher with Software Engineering Institute, East China Normal University. From July 2012 to March 2014, he was a Researcher Fellow with the National University of Singapore, Singapore. Moreover, from April 2014 to December 2014, he was a Research Scientist with Temasek Laboratory under the Ministry of Defense of Singapore. His research interests include formal method, formal modeling, and verification of real-time or control systems, and IEC 61508 and IEC 61131 standards.

Dr. Shi was the Recipient of Shanghai Science and Technology Committee Rising-Star Program, in 2018, ACM and CCF nomination of Excellent Doctor in Shanghai, in 2014.



Yueling Zhang received the B.S. degree in software engineering and the Ph.D. degree in computer science from East China Normal University, Shanghai, China, in 2012 and 2019, respectively.

She is currently working as a Research Scientist with Singapore Management University, Singapore. Her research interests include program analysis, formal modeling, and verification of artificial intelligence systems.