Towards Automated Safety Vetting of PLC Code in Real-World Plants

Mu Zhang*, Chien-Ying Chen†, Bin-Chou Kao‡, Yassine Qamsane§, Yuru Shao¶, Yikai Lin¶,
Elaine Shi*, Sibin Mohan†, Kira Barton§, James Moyne§ and Z. Morley Mao¶
*Department of Computer Science, Cornell University
†Department of Computer Science, University of Illinois at Urbana-Champaign
‡Information Trust Institute, University of Illinois at Urbana-Champaign
§Department of Mechanical Engineering, University of Michigan
¶Department of Electrical Engineering and Computer Science, University of Michigan
*mz496@cornell.edu, *elaine@cs.cornell.edu, †{cchen140,sibin}@illinois.edu, ‡ bkao2@illinois.edu,
§{yqamsane,bartonkl,moyne}@umich.edu, ¶{yurushao,yklin,zmao}@umich.edu

***Abstract*—Safety violations in programmable logic controllers (PLCs), caused either by faults or attacks, have recently garnered significant attention. However, prior efforts at PLC code vetting suffer from many drawbacks. Static analyses and verification cause significant false positives and cannot reveal specific runtime contexts. Dynamic analyses and symbolic execution, on the other hand, fail due to their inability to handle real-world PLC programs that are event-driven and timing sensitive. In this paper, we propose VETPLC, a temporal context-aware, program analysis-based approach to produce *timed event sequences* that can be used for *automatic safety vetting*. To this end, we *(a)* perform static program analysis to create *timed event causality graphs* in order to understand causal relations among events in PLC code and *(b)* mine temporal invariants from data traces collected in Industrial Control System (ICS) testbeds to quantitatively gauge temporal dependencies that are constrained by machine operations. Our VETPLC prototype has been implemented in 15K lines of code. We evaluate it on 10 real-world scenarios from two different ICS settings. Our experiments show that VETPLC outperforms state-of-the-art techniques and can generate event sequences that can be used to *automatically detect hidden safety violations*.**

I. INTRODUCTION

Industrial control systems (ICS) play an essential role in modern society. In the new era of Industry 4.0 [12], computerized control systems have become the backbone of crucial infrastructures such as power grids, transportation as well as manufacturing sectors. Compared to traditional ICS that were constructed using fixed electronic circuits, programmable logic controllers (PLC) have brought flexibility, configurability and automation to these domains. However, this freedom has also introduced complexity, and thus uncertainty, to safety-critical physical plants. Unexpected logic errors may cause serious problems such as fatal collisions or massive explosions. Reports have shown that anomalous ICS behaviors have resulted in loss of life on real-world factory floors [11], [19].

In addition, security problems are highly coupled with safety issues in the ICS domain. In fact, physical damage is one of the major goals for security breaches in ICS. Compared to attacks targeting consumers or IT systems, that often aim to make profits or steal data, cyberattacks on factory floors are intended to sabotage physical infrastructures. Real-world incidents, including Stuxnet [36], German Steel Mill Cyber Attack [49], Ukrainian Power Grid Attack [50], have shown that although adversaries must first leverage security penetration techniques to infiltrate the digital layers of modern plants, they often attempt to manipulate critical safety parameters, such as the frequency of nuclear centrifuges, and trigger benign but faulty code, to cause serious damage. Hence, there is a need for detecting situations where such safety violations can occur. Due to the complexity of contemporary ICS, that involves interactions between PLCs and various other machines, we need *automated* mechanisms to find such problems.

While there exists work [24], [28], [30], [31], [42], [44], [57], [58], [61], [63], [65] that aims to statically verify PLC logic in a formal manner, such static analysis techniques suffer from significant false positives since they are unable to reason about runtime execution contexts. For instance, they may detect potential problematic paths in the code that are infeasible at runtime. In addition, the behavior of ICS is strictly constrained by physical limits at runtime (e.g., velocity, temperature, etc.) as well as changes to these properties.

To address these limitations, prior work [35], [39], [45], [62] has explored the usage of dynamic simulations of runtime behaviors to detect PLC safety violations. In addition, recent work [43], [54] has enabled symbolic execution on PLC code. Despite their apparent effectiveness in finding bugs in independent PLC programs, these techniques are limited because they overlook an important fact that a real-world PLC is never working alone. On the contrary, it collaborates with other programmable components on the factory floor, such as robots, CNCs or even other PLCs, to carry out certain tasks. Hence, PLC logic is not only triggered by internal data inputs but also driven by external events due to the coordination and communication among multiple units. Unfortunately, the aforementioned work focuses mainly on the testing or resolution of input values and not on the complete event space of multiple collaborating components, and thus cannot automatically exercise real-life PLC programs.

To address this problem, we propose VETPLC, a temporal context-aware, program analysis-based system that *automati-*

cally constructs timed event sequences. These sequences can then enable automated dynamic safety vetting of PLC code.

Although they are still lacking in the PLC context, automated dynamic analysis and symbolic execution on event-driven programs have been well-studied in the smartphone [27], [46], [55], [67] and web [51], [66] domains. To model non-deterministic events, researchers have proposed to automatically generate event sequences of different orders, based upon program models [67] or testing [27], [46], [51], [55], [66] – to drive program execution. Yet permutation of events is insufficient to describe the conditions that lead to safety violations in PLC code. The timings, at which events are delivered, matter. This is because PLC events have implicit temporal dependencies caused by both intrinsic durations and external physical constraints. *Our key observation is that multiple event sequences of the same valid order may or may not lead to safety violations due to the different timings between events.* Thus, *generating timed event sequences* is a requisite step to successfully reveal safety issues in PLC code.

Thus, VETPLC complements the prior research on dynamic analyses and symbolic execution that search merely the value space in PLC code. It further introduces novel techniques to explore the timed event space so as to effectively exercise and examine PLC programs.

Specifically, *(a)* to uncover the order of triggering events, we first perform static program analyses on controller code (of the various interconnected units), including *PLC and robot* and generate *timed event causality graphs* to represent the temporal dependencies of cross-device events; *(b)* to quantitatively model the timing of events, we analyze the controller code to extract internal time limits, collect runtime data traces from physical ICS systems and then leverage data mining to recover temporal invariants; *(c)* combining this timing model with causality graphs, we then create timed event sequences that can serve as inputs for any dynamic PLC code analyses; to enable automated safety vetting, we formally define and manually craft safety specifications based upon expert knowledge and conduct runtime verification on PLC execution traces.

It is worth noting that previous research has also sought to create timed event sequences for testing event-driven real-time programs. Event sequences have been produced from either manually crafted specifications [48] or profiling program execution time [52]. In contrast, we automatically extract event ordering and timing using program analyses and data mining, and further enable this technique in the new domain of PLCs and broadly in the context of ICS.

To the best of our knowledge, we are the *first* to enable timing-aware safety vetting on event-driven time-constrained PLC code for real-world ICS, in particular, via extracting event temporalities from program logic and physical environments.

We have implemented VETPLC in 15K lines of code – 7K lines of C++ and 8K lines of Java. To demonstrate the efficacy of our approach, we apply it to 10 real-world scenarios on two ICS testbeds that are of completely different physical compositions: *(i)* the *SMART* [47] testbed is a scaled-down yet fully functional automotive production line and *(ii)* the *Fischertechnik* testbed replicates a consecutive part processing facility controlled by multiple collaborative PLCs. Note that the PLC programs under examination remain intact, and we did not introduce vulnerable code into them. Experimental results show that VETPLC outperforms the state-of-the-art techniques and can effectively produce event sequences that lead to deep and authentic safety bugs, which are already hidden in real-world PLC code due to developers' mistakes.

In summary, this paper makes the following contributions:

- We explore physical ICS testbeds to gain an important insight: real-world controller code is event-driven and timing-sensitive.
- We are the first to automate dynamic safety vetting of real-world PLC code via the creation of timed event sequences.
- We use custom static analyses, that address the specific programming paradigms of PLCs, to extract causal relationships among events.
- To the best of our knowledge, this is the first work that distills temporal dependencies in physical ICS testbeds.
- We have demonstrated the effectiveness of VETPLC on two different types of real-world ICS testbeds: VETPLC has found "organic" vulnerabilities in real-world testbeds.

II. BACKGROUND

Programmable Logic Controller. A programmable logic controller [18] is the core control unit of a large number of modern automation systems. It can be either used as a separated *master* controller or integrated as a *slave* controller to other machines such as CNCs. The basic functionality of a PLC is to repeatedly generate control commands based on input signals and internal control logic. On startup, a PLC is running in an infinite loop where each iteration, called a *scan cycle*, consists of three major phases. 1) *Input:* PLC reads inputs from external events (e.g., sensors) and buffers them in memory. 2) *Computation:* All variable values are fixed. The PLC then invokes its logic program and calculates new variable states based on the buffered inputs and their current states. 3) *Output:* The PLC writes the computed new states into output memory in order to start the next cycle.

PLC programming languages follow the international standard IEC 61131-3 [10]. It defines three graphical languages and two textual languages. All of the languages share IEC 61131-3 common elements and can be translated between each. In particular, the Structured Text (ST) is a high-level textual language that syntactically resembles Pascal (Figure 2) and thus is known for its understandability [20]. Notice, however, although an ST program resembles those written in other high-level languages, its dataflow is very different due to the existence of *scan cycles*. Since PLC variables are kept intact during the computation phase, value changes caused by logic code do not become effective until the next cycle. In effect, in any scan cycle, a PLC variable bears two "versions": the "current" version from the last cycle is effective at the present time; the "new" version records all the changes in the current round and eventually replaces the "current" one during

the output phase. As a result, *1) there exists no dataflow within one scan cycle; 2) dataflow happens between two neighboring cycles and the "current" value of a variable may be the result of any assignment instructions in the last cycle.*

Industrial Robot. An industrial robot is essential for performing various actuations, such as assembly, pick-and-place, packaging, etc. Robot programming languages of individual vendors are proprietary but in general fall into two categories: high-level and low-level. High-level languages, such as KAREL for FANUC robots or RAPID for ABB, are influenced by the Pascal syntax. Low-level code is assembly-like, and is developed through *teach pendants* which are handheld devices directly connected to robots. Aside from common program instructions (e.g., assignments, conditional or unconditional jumps and function calls), these programs all employ special *motion* instructions to guide physical movements and use *wait* instructions to enable delays and control timings. While Robot programs can be launched via a main function, in practice they are triggered dynamically by input events. The mapping between triggering signals and call targets is configured using teach pendants. Without loss of generality, we hereafter explain robot inner-workings based upon pick-and-place robots from FANUC that has the most industrial robots installed worldwide [56]. Specifically, we focus on its teach pendant (TP) language, depicted in Figure 8, which is the de facto standard to program FANUC robots [1].

Cross-Device Communication. A PLC and a remote device communicate via signals using industrial network protocols, such as EtherNet/IP [8]. The remote device opens multiple pins for inputs and outputs. For example, a FANUC robot can enable 512 bits of digit inputs (DI) and 512 bits of digit outputs (DO). On the PLC side, each remote pin is mapped as a base address (i.e., IP address) plus an offset. Thus, PLC code can control a remote device by directly accessing these mapped I/O bits. The I/O mappings are automatically configured when a remote device is added to an ICS environment supervised by a PLC. Once its IP address is determined, the underlying EtherNet/IP protocol takes the responsibility to recognize the I/Os on this device and bind them to PLC variables.

III. Problem Statement & Approach Overview

A. Motivating Example

We motivate our problem using our *SMART* testbed [47], depicted in Figure 1. This testbed represents a fully functional assembly line that produces model cars. It consists of a gantry crane, a circular conveyor belt, 2 pick-and-place robots, 3 CNC (Computer Numerical Control) machines, and is controlled by a PLC. Particularly, it is equipped with Allen Bradley PLC from Rockwell Automation[1] and FANUC robots[2].

It is worth noting that the *SMART* testbed is a miniature of real-world automotive manufacturing sectors. It has been established and constantly upgraded for over 20 years, and has been used for numerous projects over the decades. This testbed

[1]Leading PLC supplier in North America w/ 60% of the market share [17]
[2]The most popular industrial robots worldwide [1]

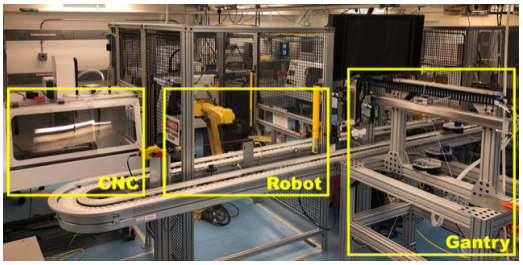


Fig. 1: SMART Testbed for Manufacturing Model Vehicles

was developed by engineers from Rockwell Automation, faculty and graduate students: the hardware components and the way they connect precisely resemble those on real-world factory floors; a large body of controller code (e.g., robot motion, CNC operation, RFID I/O, etc.) was directly borrowed from industry practices [7]. The fidelity of this control system has been verified through consistent collaboration with Rockwell Automation.

Physical Compositions. The gantry system serves as the entry and exit points of the testbed. It delivers empty pallets to CNC machine #1 to start the manufacturing processes and, eventually, it removes the produced parts from the conveyor. The circular conveyor belt is always on and keeps moving the pallets around the robots and CNCs. The robots and CNC machines are organized into two cells to accomplish different tasks (e.g., molding, flipping, etc.), where Cell 1 is comprised of Robot #1 and CNC #1, and Cell 2 contains the rest. Immediately in front of each cell are RFID transceivers that can sense the presence of incoming pallets, empty or loaded, because RFID tags are attached to both pallets and parts. The RFID tag on a part maintains a numerical value indicating its next manufacturing process. A pallet stopper is also installed to every cell to block moving pallets. By default, the stopper is always enabled to block any arriving pallets unless a signal that indicates otherwise is received.

PLC and Robot Logics. Figure 2 and Figure 8 (in Appendix A) show in part the control logic of the PLC and Robot #1 in Cell 1, respectively. The code snippets depict how a processed part is passed from CNC to conveyor.

Since a raw part has been delivered by the gantry to the CNC for processing, the PLC code (Figure 2) is now expecting to receive the processed part and deliver it to the next cell using an empty pallet. The coordination between PLC and robot is realized through events. In order to receive and send these signals, 6 input variables (Ln.3-7,52), 2 output variables (Ln.8-9) and 4 internal variables (Ln.11-13,49) are declared. In each scan cycle, the PLC first clears the output variables during initialization (Ln.16-19) and then checks all the input variables sequentially to update the outputs (Ln.21-44).

More concretely, Ln.21-23 first update the availability of an empty pallet at Cell 1 (`Pallet_Arrival`) by checking the presence of a pallet (`Pallet_Sensor`) and also the absence of a part (`NOT(Part_Sensor)`). If, however, an incoming pallet is already loaded with a part (Ln.25-27), the PLC will send a signal via `Retract_Stopper` to retract the stopper and let this pallet pass through. When an empty pallet has arrived at

```
1  PROGRAM CELL1
2    VAR
3      Pallet_Sensor AT %IX0.1 : BOOL;
4      Part_Sensor AT %IX0.2 : BOOL;
5      CNC_Part_Ready AT %IX0.3 : BOOL;
6      Robot_Ready AT %IX0.4 : BOOL; //DO[6]
7      Part_AtConveyor AT %IX0.5 : BOOL; //DO[2]
8      Retract_Stopper AT %QX0.1: BOOL;
9      Deliver_Part AT %QX0.2 : BOOL; //DI[0]
10
11     Pallet_Arrival AT %MX0.1 : BOOL;
12     Update_Part_Process AT %MX0.2 : BOOL;
13     Update_Complete AT %MX0.3 : BOOL;
14   END_VAR
15
16   Pallet_Arrival := false;
17   Retract_Stopper := false;
18   Deliver_Part := false;
19   Update_Part_Process := false;
20
21   IF Pallet_Sensor AND NOT(Part_Sensor) THEN
22     Pallet_Arrival := true;
23   END_IF;
24
25   IF Part_Sensor THEN
26     Retract_Stopper := true;
27   END_IF;
28
29   IF Pallet_Arrival AND CNC_Part_Ready AND Robot_Ready AND
         NOT(Part_AtConveyor) THEN
30     Deliver_Part := true;
31     Update_Part_Process := true;
32     CNC_Part_Ready := false;
33     Robot_Ready := false;
34   END_IF;
35
36   IF Update_Part_Process THEN
37     //Call subroutine to update process No.
38     UPDATE_PART(2);
39   END_IF;
40
41   IF Update_Complete AND Part_AtConveyor THEN
42     Retract_Stopper := true;
43     Update_Complete := false;
44   END_IF;
45 END_PROGRAM
46
47 PROGRAM UPDATE_PART
48   VAR_INPUT
49     Part_Process AT %MD50 : DWORD;
50   END_VAR
51   VAR
52     RFID_IO_Complete AT %IX0.6 : BOOL;
53     Update_Complete AT %MX0.3 : BOOL;
54   END_VAR
55   //Perform 15-step I/O operations on RFID
56   ...
57   IF RFID_IO_Complete THEN
58     Update_Complete := true;
59   END_IF
60 END_PROGRAM
```

Fig. 2: PLC ST Code for Picking Up Processed Parts

Cell 1, the PLC code (Ln.29-34) will further check the Boolean inputs, `CNC_Part_Ready`, `Robot_Ready` and `NOT(Part_AtConveyor)`, to confirm the existence of a processed part, availability of robot and clearance of parts on the conveyor, respectively. If all the conditions are satisfied, the PLC will then perform two actions: 1) requesting the robot to pass the processed part to pallet and 2) updating the manufacturing process number on the part. Two signals `Deliver_Part` and `Update_Part_Process` are thus enabled.

1)`Deliver_Part`. Based upon configuration, the variable `Deliver_Part` is mapped to a digital input (`DI[0]`) on the robot side. Being true, this signal triggers the robot program in Figure 8 to execute. The robot code then operates the robot arm, via a series of motion instructions such as linear movement ``L'' or joint movement ``J'', in order to pick up a part from the CNC machine (Figure 8 Ln.6-12) and pass it to the conveyor (Figure 8 Ln.18-20). When the part has been delivered to the conveyor, the robot turns on its output signal `DO[2]` for 0.5 seconds to indicate the completion (Figure 8 Ln.22-24). This output is then mapped to `Part_AtConveyor` on the PLC. In the end, the robot returns to a safe zone.

2)`Update_Part_Process`. When this variable is true, a subroutine `UPDATE_PART(int)` is called to conduct a 15-step I/O operation on the RFID attached to the part (Ln.36-39). When this is done, the subroutine (Ln.47-60) will receive a `RFID_IO_Complete` signal and then notify its caller by setting the Boolean variable `Update_Complete`.

To check whether the two actions are completed, PLC constantly reads two response signals `Part_AtConveyor` and `Update_Complete`. When both signals are true, PLC will retract the stopper to transfer this loaded pallet (Ln.41-44).

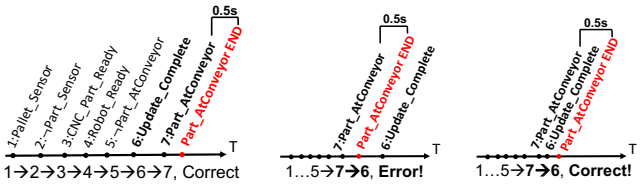
Safety Violation and Root Cause. This code, in fact, can lead to item overflow [9], which is a typical type of safety issues on the factory floor. Fundamentally, it is caused by mismatched expectations between the sender (robot) and receiver (PLC) of event `Part_AtConveyor`'s duration.

The signal `Part_AtConveyor` has dual purposes. When it is true, it indicates the robot has delivered a part to the pallet, which can now leave the cell. When it is off, that means the conveyor has been cleared to accept a new part, and the robot can then move away from conveyor for another delivery. However, in practice, the robot does not need to stop at conveyor waiting for the pallet to leave. Although the robot cannot pass the second part to the conveyor prior to the departure of first one, the robot can, in fact, move towards the CNC in advance to save time for the next delivery. For the sake of saving time, the developers implemented a timeout in the robot code and only allowed the event `Part_AtConveyor` (`DO[2]`) to last for 0.5 seconds (Figure 8 Ln.23-24), no matter if the conveyor is cleared by then. As a result, the robot is guaranteed to start handling another delivery 0.5 seconds after the previous one.

Unfortunately, if the robot turns off `Part_AtConveyor` prematurely, the PLC may never see both `Part_AtConveyor` and `Update_Complete` being set to true at the same time, either due to an unexpectedly fast part delivery or slow RFID update. This is also because PLC developers typically do not buffer old signal values (in this case, `Part_AtConveyor` being "TRUE") but rather always read data directly from their origins, in order to avoid synchronization problem.

In fact, a real-world error has been reported from the *SMART* testbed when the speed of robot is increased to a certain extent, and thus `Part_AtConveyor` ends even before the update of process number is complete. Then, there exists no window when both `Update_Complete` and `Part_AtConveyor` are true (Figure 3b). In that case, even if the pallet has already been loaded, it can never leave the cell.

This error can cause a serious safety issue since the conveyor will overflow due to the constantly arriving pallets.



(a) Sequence 1 (b) Sequence 2 (c) Sequence 3

Fig. 3: Event Sequences with Different Orders and Timings

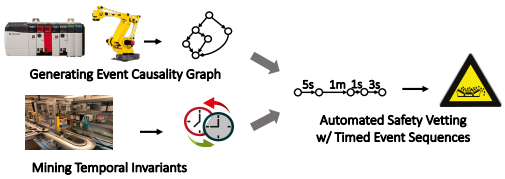


Fig. 4: Overview of VETPLC System

Eventually, it will cause pallets to collide and fall, or even cause the overloaded conveyor to break. Though seemingly straightforward, this is in fact a typical safety violation that can cause severe injuries on the factory floor and thus has attracted attention in both industrial practices [5], [6], [9] and academic research [37].

It is worth noting that although we highlight this issue using collaborative PLC and a robot, it is actually a common problem that can be caused by coordination of any types of controllers, such as multiple PLCs, PLCs and CNCs (controlled by an integrated slave PLC) or CNCs and robots. Both our experience and domain knowledge from field engineers (from Rockwell) show that a large portion of PLC safety problems originated from the coordination required between multiple units because they are manufactured by different vendors and programmed individually without considering different contexts (e.g., timing). Nevertheless, we believe the problem involving PLCs and robots is the most challenging one to address because it requires the understanding of multiple programming languages and their interactions. Hence, we focus on such a case to explain our approach. However, as we show in the evaluation, our system can be applied to other classes of coordinating systems as well.

Challenge for Detecting the Problem. Static analyses may cause significant false positives due to the lack of runtime constraints and thus cannot easily address this problem. For instance, a potential error state detected by static analysis may only be triggered when the speed of robot is greater than 10m/sec, which however can never be reached in practice.

In contrast, dynamic analysis and symbolic execution do not cause false positives. To use them on event-driven programs, prior work [27], [46], [51], [55], [66], [67] generated event sequences of different orders to exercise code and explore paths. In our case, one can create an event sequence following the order of `1:Pallet_Sensor` ⇝ `2:¬ Part_Sensor` ⇝ `3:CNC_Part_Ready` ⇝ `4:Robot_Ready` ⇝ `5:¬ Part_AtConveyor` ⇝ `6:Update_Complete` ⇝ `7:Part_AtConveyor`, as illustrated in Figure 3a. Note that eventually `Part_AtConveyor` terminates due to the robot logic. Exercising PLC code using such this sequence does not lead to any error. One can then permute the events by switching `6:Update_Complete` and `7:Part_AtConveyor` (Figure 3b). Then, the safety problem will occur at runtime.

However, just rearranging the event order may not solve the path discovery problem in time-constrained controller programs. For instance, the event sequence in Figure 3c shares the same ordering as the one in Figure 3b, yet it cannot cause the error. When the time difference between events `7` and `6` changes, the consequence may also vary.

To address this problem, we expect to automatically produce effective, error-triggering event sequences (such as Figure 3b) by considering both ordering and timing of events. Notice that an alternative approach is to model internal timeouts as external events and then perform event permutation without considering timing. For example, the termination of event `Part_AtConveyor` can then become another independent event, and the permutation thus is conducted over 8 events. However, we would argue that this solution has two major shortcomings: 1) it may drastically increase the event space; and 2) the generated sequences can cause false alarms because they may still violate critical time and physical constraints and thus are actually invalid. Its fundamental limitation lies in the fact it assumes the complete independence of individual events and does not quantitatively consider their temporal contexts.

B. Threat Model

We consider that adversaries can trigger vulnerabilities in benign (but faulty) PLC code via manipulation of configuration options that impact important physical properties such as machine speeds. In addition, we also consider that insiders can compromise PLC source code to intentionally inject (stealthy) safety violations (e.g., PLC logic bombs [41]). Note that insider attacks are top security challenges [40], [64] for air-gapped ICS and have been identified in major ICS incidents including Stuxnet and the Maroochy Water Services Attack [23]. As a result, PLC source code and configurations may not be trustworthy. Note though, we assume that the rest of the ICS environment, including hardware and operating systems, as well as our data collection mechanisms are trusted.

It is worth mentioning that, at this point, our work is mainly focusing on the detection of safety violations. However, some of the techniques we developed can also be useful to address security challenges in the ICS context.

C. System Overview

To achieve our goal, we have developed VETPLC, that consists of 3 major steps. Figure 4 illustrates its architecture. We hope to deploy VETPLC as a vetting tool to examine any PLC code before it is released for a production system.

(1) **Generating Event Causality Graph.** Given the PLC and robot code, we first perform static program analyses to extract the event causality graphs for interconnected devices. We further leverage specified I/O mapping to handle cross-device communication.

(2) **Mining Temporal Invariants.** Next, to understand those quantitative temporal relations that cannot be revealed by

program code, we collect runtime data traces of PLC variables from physical ICS testbeds. We then examine the traces to infer the occurrences of particular events and conduct data mining to discover temporal event invariants.

(3) **Automated Safety Vetting with Timed Event Sequences.** Constrained by the generated timed event causality graphs, we perform event permutations to automatically create timed event sequences. Then, we apply the generated sequences to exercise PLC code for dynamic analysis. To automatically identify safety problems, we formalize and craft safety specifications according to expert knowledge so as to perform runtime verification.

IV. Timed Event Causality Graph

A. Key Factors

A naïve approach to deriving event sequences is to consider every combination of events. For instance, prior work has presented a baseline approach, AllSeqs [27], that exhaustively permutes all UI events to create triggering sequences for testing Android apps. However, due to the massive possible permutations, such a solution can be prohibitively time consuming. In fact, not all permutations are valid sequences because the causal dependencies of PLC events are inherently constrained by controller code. To reduce the search space, we can extract such dependencies from program logics in the first place. Particularly, we are interested in three causal factors.

- **Control-Flow.** We take into account intra-procedural, inter-procedural and cross-device control flow dependencies: 1) within a function, event variables evaluated in an IF-Condition have direct causal impact on those defined in its IF-Clause; 2) for function calls, we consider that the callsite in the caller causes all the logic in the callee; 3) cross-device event exchanges via mapped I/O indicate the causal relations between code on multiple controllers.
- **Constants.** The constant value of an event-related variable in an IF-Condition can partially determine if the IF-Clause becomes effective. Thus, the dataflow from the constant assignment to the condition check of this variable indicates that the former causes the latter.
- **Event Duration.** The causal effect of events may last for a certain amount of time when subsequent states are maintained. Machines with local memory can produce events with permanent states. The PLC can also help preserve the states of transient signals (i.e., sensor readings) or its internal events. In the meantime, event senders can also proactively terminate signals based upon timing.

In addition to these internal factors, the occurrences of events are also affected by external timing constraints caused by physical actions, such as robot motion and external I/O operations. We will discuss this in Section V.

B. Formal Definition

To interpret the internal constraints on event ordering, we extract the causal and temporal relations among events from PLC and robot code to generate dependency graphs. In particular, we describe the cross-device event dependencies

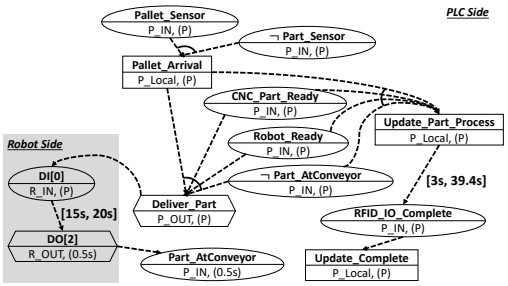


Fig. 5: The TECG of the Motivating Example

using *Timed Event Causality Graphs* (TECGs). At a high level, a TECG is based upon the And-Or Graph [53] that can illustrate the causalities among events and express their and/or relationships. A formal definition is presented as follows.

Definition 1. A *Timed Event Causality Graph* is a directed graph $G = (V, E, \alpha, \beta)$ over a set of events Σ and a set of time durations T, where:

- The set of vertices V corresponds to the events in Σ;
- The set of edges $E \subseteq V \times V$ corresponds to the *causal dependencies* between events, where the combination of all immediate predecessors of a vertex can always cause this successor event to happen. Specifically, if some of these predecessor vertices form a conjunction, their outgoing edges become compounded using an "arch"; if they form a disjunction, the corresponding edges are separated.
- The labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the labels of corresponding events, where each label is comprised of 3 elements: event name, class and duration.

An event is named after the atomic proposition it affects. For instance, if an event causes `a==15` to be true, we name it as "`a==15`"; if it causes Boolean `c` to be false, we refer to it as "¬`c`". We consider 6 classes of events, including input (P_IN), output (P_OUT), local (P_Local) events of PLC and those of a remote device (R_IN, R_OUT, R_Local). The event duration is either Permanent (P), meaning it is always enabled until turned off by PLC logic, or a finite amount of time.

- The labeling function $\beta : E \rightarrow T$ associates edges with the labels of time intervals. These labels are concrete numbers if we can retrieve the corresponding time intervals from ICS testbeds; otherwise, they are "Indeterminate".

C. TECG of Motivating Example

Figure 5 depicts the TECG of the motivating example. At first, this automation system expects to receive events from two sensors. The conjunction of a positive event, `Pallet_Sensor`, and a negative one, ¬ `Part_Sensor`, triggers the PLC local event `Pallet_Arrival`. Then, if all of the 4 events, `Pallet_Arrival`, `CNC_Part_Ready`, `Robot_Ready` and ¬ `Part_AtConveyor` are received, the PLC will signal the robot via an output event `Deliver_Part`.

Hence, the conjunction of these four events leads to the generation of `Deliver_Part`, and such a causal dependency is represented by the compounded edges from the former to the latter. Further, `Deliver_Part` is mapped to the robot event

DI[0], which causes the robot arm to function. Once its operation is completed, the robot turns on the output DO[2] and in effect sends the event Part_AtConveyor back to the PLC. Thus, these events are connected due to cross-device control dependencies. Since DO[2] (Part_AtConveyor) terminates in 0.5 seconds according to the robot code, its duration is "0.5s" instead of "Permanent".

In the meantime, when the conjunction of aforementioned 4 events is satisfied, another PLC local event Update_Part_Process will occur. This event causes a subroutine call, in which PLC starts to update the process number encoded in the RFID on the part. Once the update is done, the RFID replies to the PLC with RFID_IO_Complete, which in turn triggers Update_Complete that the main routine expects.

By default, the time intervals of all edges are "Indeterminate", and thus are not shown on this graph. We later perform data mining on traces collected from ICS testbeds to extract temporal invariants associated with certain edges, such as Update_Part_Process $\xrightarrow{[3s,39.4s]}$ RFID_IO_Complete.

D. Graph Construction

To generate TECGs, we perform static analyses that are tailored for the unique programming paradigms of PLC code.

a) Special Consideration for PLC Scan Cycles: Prior work has paid special attentions to PLC's dedicated data types, such as Timers and Counters [54], and its preemptive thread scheduling model [43]. In addition, we believe that it is also crucial to take into account PLC's *scan cycles* that cause implicit, yet significant impact, to entry points and dataflow of PLC code. Nevertheless, to the best of our knowledge, this has never been seriously explored in prior work.

Entry Point Discovery. PLC code is event-driven and thus all its event handlers are program entry points. In contrast to typical event-driven programs that use dedicated constructs to explicitly implement event handling mechanisms, event handlers in PLC code are implicitly defined using IF-Conditions. Because internal value changes in one scan cycle do not become effective until the next one begins, the IF-Conditions in PLC code can only be affected by external inputs received at the beginning of a cycle. Therefore, in effect, they act as event handlers to capture either new sensor readings or updates from last cycle. Hence, an IF-Condition becomes the entry point of its IF-Clause code as well as the subroutines called by the IF-Clause. For IF-Clause code wrapped by nested IF-Conditions, we consider the inner-most one to be its entry point.

Dataflow Analysis. The fact that variables are of fixed value in every cycle also causes the dataflow to change. As explained in Section II, the process of dataflow analysis for PLC code is mainly to track data dependencies between scan cycles. Further, due to the existence of asynchronous event handlers, the analysis should compute data reachability from any "define" in one cycle to any "use" in the next.

b) Graph Construction Algorithm: Our algorithm for generating timed event causality graphs is illustrated in Algorithm 1. This algorithm expects to receive three inputs, PLC, $\mathbb{REMOTE}$ and IOMapping. They represent PLC code, a set of remote controller code (e.g., robot code) and the I/O mappings between PLC and remote devices, respectively. Its output is a timed event causality graph, TECG, which is comprised of a set of edges. The I/O mappings are automatically established when remote devices are added to the PLC and thus can be retrieved from PLC configurations.

During initialization, we set TECG to be an empty set. Next, we transform all predicates in the IF-Conditions of PLC code into disjunctive normal form (DNF) in order to illustrate them using an And-Or graph. Thus, an original predicate becomes a set of sub-predicates connected via "OR" logic, while each sub-predicate is a conjunction of events depicted as compounded edges. Further, we retrieve all the entry points (i.e., IF-Conditions) $\mathbb{EP}$ of PLC code. Meanwhile, we also link neighbors of nested IF-Conditions to show their control relations. Then, we iterate over every event (i.e., atomic proposition) pin in $\mathbb{EP}$ and seek its root causes, which are events or event combinations that can always lead to pin.

We first aim to discover the root causes for pin within the PLC code. To this end, we perform use-def chain analysis to obtain the definition set $\mathbb{DEF}$ of pin and then look for the entry point EP (again, IF-Conditions) of each definition def in $\mathbb{DEF}$. The events in EP thus have causal impact on def and on pin. To ensure the positive causal dependency between EP and pin, we also conduct constant analysis for def. If def is a constant and its value can satisfy pin, we can then determine that EP can cause pin to happen. Hence, we call TECG.AddCompoundEdges() to link EP with pin and handle the construction of compounded edges.

It is worth noting that since IF-Conditions in one scan cycle can be affected by any code in the previous one (dataflow-wise), our use-def chain and constant analyses will look for definitions from everywhere in PLC code. Ideally, we can consider an infinite chain of scan cycles and compute backward dataflow exhaustively in an iterative fashion. However, such computation is excessively expensive. Besides, the generated dependencies can be extremely complex (e.g., conditional dependencies) and therefore may not be easily applied to event sequence generation. Thus, in practice, we take a conservative approach and only look back for one previous cycle. As a result, our analysis may miss some dependencies in specific conditions. Nevertheless, while missing a dependency may lead to invalid permutations of events, it does not result in the exclusion of valid event sequences. Moreover, our evaluation shows that, although conservative, our analysis can already help remove a large number of invalid sequences.

Besides searching for intra-PLC causalities, we also seek possible root causes of pin across devices. Our cross-device analysis starts from Ln.13. It is performed on an on-demand basis and only begins when pin is mapped to an output of a remote device. If pin indeed exists in the IOMapping, we retrieve its mapped counterpart $rout$ and add an edge $(rout, pin)$ into TECG. Then, we search for the entry point REP for $rout$ in the code of remote controller (e.g., robot, CNC, PLC). The entry point REP represents the trigger of $rout$. If any input rin in REP can be mapped to a PLC output

Algorithm 1 Construction of Timed Event Causality Graph

```
1: procedure BUILDTECG(PLC, ℝEMOTE, IOMapping)
2:     TECG ← ∅
3:     TRANSFROMPREDICATESTODNF(PLC)
4:     𝔼P ← GETANDLINKENTRYPOINTS(PLC)
5:     for ∀pin ∈ 𝔼P do
6:         𝔻EF ← USEDEFCHAIN(PLC, pin)
7:         for ∀def ∈ 𝔻EF do
8:             if ISCONST(def) ∧ ISSATISFIED(pin, def) then
9:                 EP ← GETENTRYPOINT(PLC, def)
10:                TECG.ADDCOMPOUNDEDGES(EP, pin)
11:            end if
12:        end for
13:        if IOMapping.EXISTS(pin) then
14:            rout ← IOMapping.GET(pin)
15:            TECG ← TECG ∪ (rout, pin)
16:            REP ← GETENTRYPOINT(ℝEMOTE, rout)
17:            for ∀rin ∈ REP do
18:                if IOMapping.EXISTS(rin) then
19:                    pout ← IOMapping.GET(rin)
20:                    TECG ← TECG ∪ (pout, rin)
21:                    EP ← GETENTRYPOINT(PLC, pout)
22:                    TECG.ADDCOMPOUNDEDGES(EP, pout)
23:                end if
24:            end for
25:        end if
26:    end for
27:    ADDEVENTCLASSANDDURATION(TECG, PLC, ℝEMOTE)
28:    return TECG
29: end procedure
```

$pout$, the edge $(pout, rin)$ will be added to TECG as well. We then trace back from $pout$ to find its entry point EP in PLC code, and add compounded edges from EP to $pout$.

The last step for graph construction is to annotate vertices with event classes and durations. Event classes can be explicitly obtained from the variable declarations in PLC/CNC code or robot specifications. The durations of all events by default are set to be "Permanent" (P). Only if we can infer the concrete time duration of an event, will we safely update its label. To this end, for each input event (i.e., atomic proposition), we first discover the constant definitions that cause the proposition to be true. Then, we discover all the negative redefinitions that lead the proposition to be false. Next, we perform intra-procedural reachability analysis from the definitions to those redefinitions. If a reachable path is discovered, we further examine every statement along the path to see if any time-related instructions (i.e., wait) are present. If so, we extract and accumulate their constant parameters as the duration of this event. We do not handle variable parameters in this work.

The implementation is further explained in Appendix B.

V. Discovery of Temporal Context

A. Data Collection

Collecting Data Instead of Events. Ideally, we hope to directly collect event traces from ICS testbeds to identify their temporal behavior. However, this requires instrumentation of various distributed data sources, including sensors, robot I/O modules, RFID, etc. and therefore is an extremely difficult and tedious task. On the contrary, the data trace of PLC variables is easier to obtain due to standardized communication protocols. Yet it only preserves the runtime states of these variables but does not record the events that cause the states to transition. To bridge this gap, we intend to infer the presence of events based upon value changes in data traces and thus manage to approximate the collection of discrete physical events with the retrieval of continuous data traces.

Interesting Properties. We are interested in three properties of PLC variables: name, value and timestamp. Variable name serves as the unique identifier of a variable; the instant value of a variable reflects its current state and can be affected by specific events; the timestamp is the system time when the variable is being observed. Thus, we can define a data item d in our observation as a triple: $d = (var\_name, value, time)$.

Querying Realtime Data in Recurring Operations. We collect both positive and negative data traces from running testbeds. A positive instance begins with the arrival of empty pallet and ends in the successful departure of a loaded pallet, and thus contains all the interesting stages such as robot delivery and RFID update. A negative instance does not lead to the successful stage due to multiple reasons, such as arriving pallet loaded with part, robot not ready, CNC not ready, etc. For every instance, we keep logging all the variable values over time in order to retrieve runtime data traces. Formally, a data trace DT is a list of data item d: $DT = \{d_0, d_1, ..., d_n\}$. In practice, we run Cell-1 logic 20 times and collect 10 positive and 10 negative instances, each of which takes approximately 25 minutes. Thus, our dataset consists of a set of data traces and we refer to it as: $\mathbb{DT} = \{DT_0, DT_1, ..., DT_m\}$, where $m = 19$. We obtained 1.2 GB data in 10 hours from our testbed that runs logic code containing 35 variables.

It is noteworthy that, although limited, our dataset in practice can already help reveal the necessary invariants for detecting real-world safety problems. One possible solution to increase the amount and diversity of data traces is to follow a state-of-the-art technique (i.e., code mutation [33]) and automatically produce a large quantity of positive and negative data traces to cover a majority of normal and abnormal cases. We leave the systematic trace construction as future work.

B. Mining Temporal Properties

Inferring Discrete Events from Data Traces. For each data trace DT_i in our dataset $\mathbb{DT}$, we need to first infer the existence of events. To this end, we first divide every DT_i into multiple sublists $\{DT_i^{v_0}, DT_i^{v_1}, ..., DT_i^{v_k}\}$ where items in an individual list share the same variable name. We then iterate over each sublist. If we discover a difference between values of two neighboring items d'_l and d'_{l+1}, we record a new event $e = (type, time)$, where the $type$ is denoted using the new state of this variable and the $time$ is the timestamp of d'_{l+1}. For instance, if the value of variable `Deliver_Part` rises from 0 to 1 at time 33, then we identify an event (`Deliver_Part`, 33); if `Part_AtConveyor`'s value drops from 1 to 0 at time 60, then we find an event (¬ `Part_AtConveyor`, 60). Eventually, we merge discovered events from all sublists and thus convert a data trace DT_i into an event trace $ET_i = \{e_0, e_1, ..., e_p\}$. We therefore obtain a dataset of event traces $\mathbb{ET} = \{ET_0, ET_1, ..., ET_{19}\}$. The formal algorithm is presented as Algorithm 3 in Appendix C.

TABLE I: Mined Invariants

Event Pair	Invariant
□(Deliver_Part → ◊Part_AtConveyor)	[24.4s, 24.6s]
□(Update_Part_Process → ◊RFID_IO_Complete)	[15s, 20s]
□(Update_Part_Process → ◊Update_Complete)	[15s, 20s]

Temporal Invariants for Events. Once we have generated event traces, we would like to uncover constant time intervals between events of different types. Such constants can reflect the operation time of specific machines. However, in reality, due to the variation in program paths and indeterminism of mechanical, physical or chemical processes, the durations of real-world machine operations are never constant. On the other hand, due to physical and logical limits, machine actions are bounded by time constraints. Hence, our goal is to identify such "soft" invariants of event temporalities that fall into specific ranges. We formally define temporal invariants using Timed Propositional Temporal Logic (TPTL) [26]:

Definition 2. Let ϵ_a and ϵ_b be two event types. Then a *temporal invariant* is a property that relates ϵ_a and ϵ_b in both of the two following ways:

$\Box t_x.(\epsilon_a \rightarrow \Diamond t_y.(\epsilon_b \wedge t_y - t_x \geq \tau_{lower}))$: In an event trace, if an event instance of type ϵ_a occurs at time t_x, then another of ϵ_b eventually will happen in the same trace at a later time t_y, while the time difference between t_y and t_x is at least τ_{lower}.

$\Box t_x.(\epsilon_a \rightarrow \Diamond t_y.(\epsilon_b \wedge t_y - t_x \leq \tau_{upper}))$: In an event trace, if an event instance of type ϵ_a occurs at time t_x, then another of ϵ_b eventually will happen in the same trace at a later time t_y, while the time difference between t_y and t_x is at most τ_{upper}.

As a result, a temporal invariant describes not only the order of two event types but also the lower and upper bounds of their time difference. To extract these invariants, we follow the approach in prior work (Synoptic [29] and Perfume [60]) to perform qualitative and quantitative data mining consecutively. However, unlike previous techniques that attempt to mine all possible correlations between any two events, our mining is selective and is guided by the generated TECG. Specifically, we do not need to learn certain temporal relationships for a pair of event types if they contradict the dependencies in the graph. For example, in our motivating case, since we know the temporal logic $\Box$(RFID_IO_Complete → ◊Update_Complete) holds, we do not further seek the possibility of whether Update_Complete is followed by RFID_IO_Complete.

For all the pairwise relationships of two event types, ϵ_a and ϵ_b, that do not contradict those in TECG, we first check if their qualitative temporality $\Box(\epsilon_a \rightarrow \Diamond\epsilon_b)$ holds. This is equivalent to checking if:

$$Follows[\epsilon_a][\epsilon_b] = Occurrence[\epsilon_a] \quad (1)$$

where $Follows[\epsilon_a][\epsilon_b]$ counts, in a trace, the number of type ϵ_a events followed by at least one of the type ϵ_b events and $Occurrence[\epsilon_a]$ counts the number of event instances of ϵ_a.

Once we have determined the "followed by" relationship between two event types, we use the Perfume [60] algorithm to perform quantitative mining and extract the lower and upper bounds of time differences. In the end, we discovered 3 invariants for the motivational case as listed in Table I.

Speed Reconfiguration of Real-world Machines. The mined bounds of "soft" invariants, τ_{lower} and τ_{upper}, reflect the variation in program executions and production processes. However, such bounds are still associated with pre-configured speeds of physical machines, which often times do not reach the specified hard limits. To further understand the possible impact caused by speed reconfiguration, we need to consider absolute time bounds for these machine operations.

Let job be the number of machine operations and v_{conf} be the pre-configured speed, then $\tau_{lower} \leq job/v_{conf} \leq \tau_{upper}$. To derive the absolute lower bound for the time cost t_{job}, we consider the rated motor speed v_{rated} and thus have: $(\tau_{lower} \times v_{conf})/v_{rated} \leq job/v_{rated} \leq t_{job}$.

Meantime, since the minimum machine speed theoretically can be 0, the absolute maximum time to complete a task is infinity. However, in reality, for a high throughput, machines are expected to finish jobs as quickly as possible. Thus, ideally, machines always operate at their highest speeds. Nevertheless, safety standards have been made to regulate the maximum machine speed. For instance, the American National Standards Institute (ANSI) has published ANSI RIA R15.06 [22] for Robot and Robot System Safety which recommends that robot speed should not exceed 10 in/sec (250 mm/sec) for safety-critical operations. Such recommendations can be considered as the lowest machine speeds that can guarantee efficient and safe production. With this required safety speed, v_{safe}, we can further obtain the practical upper bound of t_{job}:

$$(\tau_{lower} \times v_{conf})/v_{rated} \leq t_{job} \leq (\tau_{upper} \times v_{conf})/v_{safe} \quad (2)$$

Admittedly, to incorporate hardware limits, we need to understand the semantics of mined invariants in order to associate this additional information to correct edges. We currently address this problem using human knowledge and leave the automatic inference of event semantics as future work. With domain knowledge, we know the time for our robot to pass a part equals the time difference between Delivery_Part and Part_AtConveyor. Plus, our robot is running at 400mm/sec on average and its rated speed is 3300mm/sec. Thus, we can obtain an enhanced invariant for this event pair: [3s, 39.4s].

Enhancing TECG with Temporal Invariants. Extracted temporal invariants are then provided to the TECG. Note that they not only offer quantitative information to enhance the existing temporal relations in the graph but may also introduce new temporal dependencies. This is because the code we analyze represents only a partial view of the entire ICS environment and therefore does not contain all the event relations. As a complement, mining runtime data traces offers a holistic view of the plant and can further uncover implicit dependencies hidden from controller code.

VI. Safety Vetting with Timed Event Sequences

A. Timed Event Sequences

Once we have constructed the TECG, we can generate event sequences based upon this graph. The major challenge is how to create event permutations that conform to the quantitative dependencies illustrated by TECG. Generally speaking, to encode the mined time range of an event (i.e., "soft" temporal invariant) into a sequence, we discretize the continuous range

Algorithm 2 Generation of Timed Event Sequences

```
1: procedure BUILDTSEQS(TECG_in, ρ)
2:     Set_event ← GETEVENTSET(TECG_in)
3:     Set'_event ← DISCRETIZE(Set_event, ρ)
4:     𝕊𝔼ℚ ← PERMUTE(Set'_event)
5:     for ∀SEQ ∈ 𝕊𝔼ℚ do
6:         for ∀ev ∈ SEQ do
7:             Path ← FINDALLSOLUTIONS(TECG_in, ev)
8:             if ∄path ∈ Path : path ⊆ SEQ.SUBSEQ(0, ev) then
9:                 𝕊𝔼ℚ ← 𝕊𝔼ℚ − SEQ
10:            end if
11:        end for
12:    end for
13:    return 𝕊𝔼ℚ
14: end procedure
```

to multiple time slices and introduce a versioned event for each slice to represent its possible occurrences. To reflect the qualitative relations among events, we check every possible permutation against the graph, so as to guarantee the prerequisite for each event happens before its occurrence.

Our algorithm BUILDTSEQS is presented in Algorithm 2. It takes two arguments. The first one is $\texttt{TECG}_{\texttt{in}}$, a reduced version of TECG, which preserves solely the nodes that are PLC inputs. These input events are the necessary ones to exercise the PLC code. The second argument ρ is the discretization parameter that indicates the number of slices every time duration is divided into. On startup, our algorithm first retrieves all the events in the graph $\texttt{TECG}_{\texttt{in}}$ to generate an event set $\texttt{Set}_{\texttt{event}}$. Next, for any event in $\texttt{Set}_{\texttt{event}}$, whose starting time is within a certain range (i.e., its incoming edge is labeled with an invariant), the range is discretized using ρ to create multiple versioned events. We then replace the original event with a set of versioned ones. For instance, since `Part_AtConveyor` is enabled 3 to 39.4 seconds after `Deliver_Part`, it is discretized to be a set $\{\texttt{P\_AC}_{T+3}, \texttt{P\_AC}_{T+10}, \texttt{P\_AC}_{T+18}, \texttt{P\_AC}_{T+25}, \texttt{P\_AC}_{T+32}, \texttt{P\_AC}_{T+39}\}$ when ρ is 5.

Hence, we extend $\texttt{Set}_{\texttt{event}}$ to be a new set $\texttt{Set}'_{\texttt{event}}$. Then, we permute all the events in $\texttt{Set}'_{\texttt{event}}$ to create sequences. Notice that in every permutation, only one versioned event from the same set can be chosen. The result of this PERMUTE is a set $\mathbb{SEQ}$ containing all candidate sequences. We further check each candidate SEQ to see if it contradicts the causalities indicated by $\texttt{TECG}_{\texttt{in}}$, and if so, it will be discarded. To do so, we iterate over each event ev in a sequence SEQ, and find all the "solutions" for ev on its hosting and-or graph $\texttt{TECG}_{\texttt{in}}$. A solution for ev is a path, from ev to a top-level vertex, which includes all of its prerequisites that are required to cause ev to happen. If any solution path is covered by the subsequence from the first element of SEQ to ev, we keep this candidate SEQ. Otherwise, it is removed from $\mathbb{SEQ}$. Finally, we output the result $\mathbb{SEQ}$ as the generated timed event sequences.

For our motivating example, we can create a timed sequence, `1:Pallet_Sensor` $\rightsquigarrow$ `2:`$\neg$ `Part_Sensor` $\rightsquigarrow$ `3:CNC_Part_Ready` $\rightsquigarrow$ `4:Robot_Ready` $\rightsquigarrow$ `5:`$\neg$ `Part_AtConveyor` $\rightsquigarrow$ $\texttt{6:Part\_AtConveyor}_{T+10}$ $\rightsquigarrow$ $\texttt{7:RFID\_IO\_Complete}_{T+20}$, which can lead to the safety violation due to premature termination of $\texttt{6:Part\_AtConveyor}_{T+10}$. Detailed implementation can be found in Appendix D.

Selection of ρ. A naïve way for discretizing a time range is to merely consider its lower and upper bounds (i.e., ρ = 1). Theoretically, it is sufficient to detect the possible presence of timing-related safety violations. However, this is too coarse-grained and can only tell if an error will occur when a machine operates at its maximum or minimum speed. On the contrary, it is in fact crucial to understand the range of machine speeds that can lead to errors. Such contextual evidence can help security investigators draw a better conclusion whether a logic error is caused by attacks. For example, prior work [38] has correlated the narrowness of an error trigger with its malice. Thus, ideally, we expect to always select a larger ρ. However, the increase in time slices also leads to the growth of total number of permutations. To understand how to strike a balance, we have an empirical study in the evaluation. Nevertheless, it is noteworthy that, while a better ρ can provide informative evidence with lower cost, the selection of ρ does not affect whether we can detect a safety defect.

B. Safety Specification

The event sequences that we generate can facilitate automated path exploration for testing PLC code. However, the fact that we can reach an unsafe state does not necessarily mean we can automatically detect the problem. To enable automated detection, we need to further specify certain safety rules and programmatically verify them at runtime.

Prior work [54] has adopted linear temporal logic (LTL) to formally define safety requirements for ICSs. However, at runtime, it is hard to enforce an LTL-based rule which requires an activity to be followed by another (e.g., overflow avoidance), because the absence of a required event during limited test time does not suggest its absence at a later time. Although, in practice, these required actions must be accomplished within a certain amount of time, LTL however is not capable of describing such temporal relations in a quantitative fashion. To address this limitation, we again use TPTL [26] to quantitatively express safety specifications.

Definition 3. Let P be a set of atomic logical proposition symbols about the system $\{p_1, p_2, ...p_{|A|}\}$, e.g., sensor `Pallet_Sensor` is on, and let $\Sigma = 2^A$ be a finite alphabet composed of these propositions. Then, the set of **TPTL-based Safety Requirements** is inductively defined by the grammar:

$\pi := x + c \mid c$

$\phi := p|\pi_1 \leq \pi_2|\pi_1 \equiv_d \pi_2|false|\phi_1 \rightarrow \phi_2|\bigcirc\phi|\phi_1 U \phi_2|x.\phi$

The grammar of TPTL is further explained in Appendix E. Table II demonstrates 5 typical classes of safety specifications, which have been studied by previous academic work or required by OSHA (Occupational Safety and Health Administration). We categorize the policies based on the root causes of industrial hazards. First, a majority of safety incidents are caused by dangerous machine-machine interactions, including machine collision, machines facing overflow or underflow due to upstream machines. Second, failure to separate humans from life-threatening machines may result in fatal accidents. Last but not least, individual machines, even without interac-

TABLE II: Categories of Safety Specifications

Typical Hazard	Example Specification to Avoid Hazard	Formal Definition	References
Collision	Whenever conveyor belt starts running, a robot arm cannot come down to pick up items.	$\square(\texttt{Conveyor\_Running} \rightarrow \neg\Diamond\texttt{Robot\_Pickup})$	TSV [54]
Overflow	Once a pallet enters a cell, the stopper must be retracted within 30 seconds to release it.	$\square t_x.(\texttt{Pallet} \rightarrow \Diamond t_y.(\texttt{Retract} \wedge t_y - t_x \leq 30\texttt{s}))$	Motivating Example
Underflow	When water purification starts, water level of tanks must not below L.	$\square(\texttt{Purify\_Start} \rightarrow \neg\Diamond(water\_level \leq L))$	Chen et al. [33]
Non-Separation	When the gate for robot is opened, robot must stop working.	$\square(\texttt{Gate\_Open} \rightarrow \neg\Diamond\texttt{Robot\_On})$	OSHA Instr. [59]
Danger Zone	Upon start, the frequency of a motor in a nuclear centrifuge is between 807 and 1210 Hz.	$\square(\texttt{Start} \rightarrow \square(807Hz \leq speed \leq 1210Hz))$	Stuxnet Dossier [36]

tion with any other entities, can still result in critical damage because they operate spatially or temporally in unsafe zones.

C. Trace-based Verification.

We carry out runtime verification based upon execution traces of PLC code. Note that, while in our testbeds, all controllers (i.e., for PLCs, robots, CNCs) can physically operate and thus produce real events, in our simulations, we only analyze PLC code while modeling and simulating the inputs (i.e., events) from remote devices.

Particularly, we first run a PLC program repeatedly, while each time we exercise the code using an individual event sequence. To this end, we convert PLC ST programs into C code using the MATIEC compiler [13] and then utilize a PLC simulator [14] to execute the code. To produce execution traces, we further instrument the generated C code to dump all instructions and variable values that originated from PLC code. In the end, we conduct runtime verification for TPTL specifications on the traces. In theory, we can follow a prior approach [32] to perform comprehensive interpretation and translation of TPTL languages. However, since our safety specifications are defined at a high level and usually straightforward, thus, in practice, our runtime monitor only focuses on this small subset that we use to describe safety requirements.

VII. Evaluation

A. Experimental Setup

To evaluate the effectiveness and efficiency of our approach, we follow the methodology of previous studies [33], [43], [54] to test VetPLC on different PLC programs. However, in contrast to prior work that experimented on either synthesized PLC code without necessary physical contexts [43] or simple, isolated logic without machine interactions (e.g., traffic lights) [54], we apply VetPLC to *real-world PLC programs that are tightly coupled with specific scenarios involving interconnected physical devices*. To further demonstrate the generality of VetPLC, unlike Chen et al.'s work [33] that focused on only one particular testbed, we hope to evaluate our system on multiple scenarios for different ICS settings.

This, however, is a challenging task because it requires a deep understanding of both physical and logical domains of real-world control systems. Nevertheless, we developed 10 scenarios on two realistic testbeds, *SMART* and *Fischertechnik*, that have completely different physical compositions. The *SMART* testbed has been introduced in Section III. The *Fischertechnik* testbed (Figure 10) is a miniature that emulates consecutive processing of parts. It connects 4 cells and 2 push rams using multiple conveyors and sensors, while each cell consists of a PLC and a CNC machine. Interested readers can refer to Appendix F to learn more details about this testbed.

Table III lists the 10 scenarios from these two testbeds. We perform causality graph generation, invariant mining, event sequence construction and safety vetting on them. Our experiments have been conducted on a test machine equipped with Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz and 16GB of physical memory. The OS is Ubuntu 16.04.4 LTS (64bit).

B. Result Overview

To show the effectiveness of VetPLC, we would like to carry out comparative experiments. Unfortunately, existing work on PLC vetting, such as TSV [54] or SymPLC [43], cannot generate event sequences to automatically analyze real-world event-driven PLC code. Nevertheless, these state-of-the-art analyzers can always be enhanced to handle event-driven code if they adopt AllSeqs [27] to calculate all possible event permutations. Therefore, we implement an AllSeqs-based baseline safety analyzer for the comparison purpose.

We apply VetPLC and the baseline analyzer to our 10 scenarios, and study 3 methods that create event sequences: 1) the baseline (AllSeqs), 2) using VetPLC to generate untimed event sequences (VetPLC-Seqs), and 3) applying VetPLC to timed sequence generation. When creating timed sequences, we select three different discretization parameters, ρ = 2 (VetPLC-TSeqs-2), ρ = 5 (VetPLC-TSeqs-5) and ρ = 10 (VetPLC-TSeqs-10). Figure 6 depicts the number of sequences each method creates, while Table IV demonstrates whether generated event sequences can lead to the discovery of safety violations. Further, for safety-related errors triggered by timed event sequences, the table *also shows the ranges of corresponding machine speeds that can cause the problem*.

As shown in the table, pure ordering-based event permutations, AllSeqs and VetPLC-Seqs, cannot lead to the hidden safety violations in timing-sensitive PLC code. We do observe, from Figure 6, a dramatic decrease (up to 96%) of event permutations for VetPLC-Seqs (green curve) compared to AllSeqs (red curve). Although the decline of possible event sequences results in much less analysis runtime overhead, it does not affect whether a violation can be detected in our cases. However, provided that a timing-insensitive safety problem can be detected by AllSeqs, VetPLC can achieve it two orders of magnitude faster.

In contrast, all the timed event sequences can result in safety problems. In fact, some of the error cases, such as conveyor overflow and frozen robots, can in fact be observed occasionally from our testbeds during daily work but cannot be easily diagnosed manually. VetPLC not only helps uncover their root causes but also finds other, previously unknown, problems. Although the vulnerabilities detected in our work all originate from human mistakes, it is also possible for insiders to actively inject safety faults into PLC source code. Note that, however, VetPLC can detect any safety violations in

TABLE III: Scenarios of Safety Violations

#	Scenario Name	Testbed	Description of Hazard	Safety Specification to Avoid Hazard
1	Conveyor Overflow #1	*SMART*	Motivating Example. See Section III	$\square t_x.(\texttt{Pallet} \rightarrow \Diamond t_y.(\texttt{Retract\_Stopper} \wedge t_y - t_x \leq 30\texttt{s}))$
2	Robot in Danger Zone	*SMART*	Robot fails to return its safe zone.	$\square t_x.(\neg\texttt{Safe\_Zone} \rightarrow \Diamond t_y.(\texttt{Safe\_Zone} \wedge t_y - t_x \leq 60\texttt{s}))$
3	Conveyor Overflow #2	*SMART*	Robot stops processing parts from conveyor due to signal conflicts.	$\square t_x.(\texttt{Pallet} \rightarrow \Diamond t_y.(\texttt{Retract\_Stopper} \wedge t_y - t_x \leq 30\texttt{s}))$
4	Part-Gate Collision	*SMART*	A pallet collides with a closed gate.	$\square(\texttt{Pallet\_AtGate} \rightarrow \square\texttt{Gate\_Open})$
5	CNC Overflow	*SMART*	CNC stops processing parts from gantry due to missing signals.	$\square t_x.(\texttt{Part\_In} \rightarrow \Diamond t_y.(\texttt{Part\_Out} \wedge t_y - t_x \leq 5\texttt{m}))$
6	Ram-Part Collision	*Fischer.*	A ram starts pushing when a part has not fully entered the ram.	$\square(\texttt{Part\_Entering} \rightarrow \neg\Diamond\texttt{Ram\_Push})$
7	CNC-Part Collision	*Fischer.*	A part is passed to CNC when a preceding part is not fully discharged.	$\square(\texttt{CNC\_Busy} \rightarrow \neg\Diamond\texttt{Part\_Arrival})$
8	Conveyor Overflow #3	*Fischer.*	Parts are pushed to conveyor prematurely.	$\square t_x.(\texttt{Part\_Arrival} \rightarrow \Diamond t_y.(\texttt{Part\_Arrival} \wedge t_y - t_x \leq 6\texttt{s}))$
9	Conveyor Underflow	*Fischer.*	A conveyor belt halts operation.	$\square t_x.(\texttt{Part\_Arrival} \rightarrow \Diamond t_y.(\texttt{Part\_Arrival} \wedge t_y - t_x \geq 8.5\texttt{s}))$
10	Ram-Part Collision #2	*Fischer.*	Ram1 pushes a part to unprepared Ram2.	$\square(\texttt{Part\_Entering} \rightarrow \square\texttt{Ram\_Ready})$

PLC source code, regardless of whether they are introduced by developers or malicious logic injected by insiders.

In addition, we notice that a finer-grained time discretization may lead to a more precise error-triggering (speed range) constraints. For instance, for Scenario #8, the sequences produced by VETPLC-TSEQS-5 reveal that a push ram at speeds from 1714 to 2000 rpm can cause errors, while those of VETPLC-TSEQS-2 only indicate that it malfunctions at the minimum speed of 1714 rpm. Some cases, such as Scenario #7, may include multiple machines with variable speeds, and thus we compute the error-triggering ranges individually.

Nevertheless, the precision improvement of speed ranges comes at a price. As we discretize time into more factions, the amount of event sequences also grows significantly. Figure 6 illustrates that, compared to ALLSEQS, VETPLC-TSEQS-2, VETPLC-TSEQS-5 and VETPLC-TSEQS-10 on average yield 38%, 93% and 226% of sequences, respectively. Nonetheless, the increase of time fractions does not always lead to an improvement of error ranges. The difference between TSEQS-5 and TSEQS-10 is not as significant as that between TSEQS-2 and TSEQS-5. Yet the increase of permutations for TSEQS-10 is drastic. As a result, empirically, we can see that TSEQS-5 strikes a balance between efficiency and precision.

C. Case Study

We perform case studies on two scenarios. The study on Scenario #2 is presented here while the study on Scenario #7 is elaborated in Appendix G.

Scenario Description. Scenario #2 depicts the interaction among a PLC, a robot and a CNC in Cell 2. Here, the robot carries a part into CNC cabinet, places it on CNC table and moves out. It then pauses at a temporary position and waits for further instructions from PLC. Normally, CNC senses a part's arrival from its table and notifies the PLC of the receipt. Then, the PLC signals the robot, allowing it to return to its safe zone, while the CNC begins to process the part.

Timed Event Causality Graph. Figure 7 illustrates the TECG constructed from PLC, robot and CNC (slave PLC) code. The causal relation between `Deliver_Part_to_CNC` and `Part_Delivered` indicates the request and response between PLC and robot. The duration of `Part_Present` extracted from CNC code is 1 second. However, the controller code cannot reveal the implicit relation between PLC sending a request to robot and CNC receiving a part, because the PLC does not directly send commands to the CNC. Fortunately, VETPLC can recover this dependency via invariant mining and thus introduce a new edge `Deliver_Part_to_CNC` $\rightarrow$ `Part_AtTable`, depicted by the bold line. Besides, data mining also discovers the robot delivery time, corresponding to `Robot_Start` $\xrightarrow{[0.5s,6.6s]}$ `Robot_Standby`.

Automated Safety Vetting. TECG helps reduce the amount of possible event permutations from 13700 to 446. We further obtain 2366, 8846 and 29246 timed sequences for TSEQS-2, TSEQS-5, TSEQS-10, respectively. Using these timed sequences to exercise the PLC code, we discover a safety violation that the robot, running at certain speeds, cannot return to its safe zone. Particularly, TSEQS-5 can provide a relatively precise error-triggering range [250 mm/sec, 959 mm/sec] with a relatively low time cost (8846 permutations).

Root Cause. This problem is caused by event timings and thus is not revealed by ordering-based sequences. Since `Part_Present` only lasts for 1 second, when PLC receives `Part_Delivered` from the robot, the former event may have already terminated. Then, PLC will not permit the robot to move back due to missing necessary signals. Such a problem can only be observed when the robot speed falls into the discovered range.

Security Implication. Our analysis results do not automatically infer the intent of safety violations, but they do serve as contextual evidence that can help investigators draw correct conclusions. Prior work [38] has indicated that attacks are likely to be triggered under very narrow conditions (e.g., logic bombs) to evade detection; Stuxnet [36] code injected by insiders runs only when the target system operates between 807 Hz and 1210 Hz – a unique frequency range used for nuclear centrifuges. Hence, if the vulnerabilities are injected by insiders, VETPLC must find their narrow triggering ranges. Otherwise, we must not provide a misleading result implying the error can happen only when robot runs at very low speed [250 mm/s, 465 mm/s] or its highest speed 3300 mm/s. Instead, we must discover a precise error-trigger range, e.g., [250 mm/s, 959 mm/s] for robot speed.

D. Runtime Performance

It takes on average 203s to construct graphs for one scenario. The computation time is acceptable because our analyses are designed to be straightforward and real-world PLC code is not very complex. The runtime of trace-based verification is proportional to the number of testing sequences, and thus is comparable to that of ALLSEQS, while each run takes approximately 55 seconds.

VIII. DISCUSSION

Scalability. Our testbeds are smaller in size, but they accurately represent certain plants that manufacture specific products. For instance, a small-scale plant, such as an aircraft seating factory consisting of 20 CNCs, often organizes its

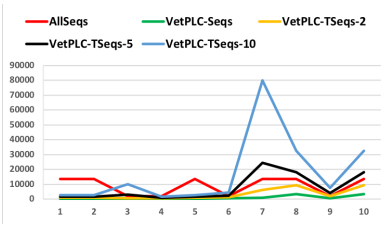


Fig. 6: No. of Event Sequences

#	AllSeqs	VetPLC-Seqs	VetPLC-TSeqs-2		VetPLC-TSeqs-5		VetPLC-TSeqs-10	
1	N	N	Y	Robot:[**3300**,3300]	Y	Robot:[**550**,3300]	Y	Robot:[**550**,3300]
2	N	N	Y	Robot:[250,**465**]	Y	Robot:[250,**959**]	Y	Robot:[250,**1486**]
3	N	N	Y	Robot:[**465**,**465**]	Y	Robot:[**307**,**959**]	Y	Robot:[**275**,**1486**]
4	N	N	Y	Robot:[250,**467**]	Y	Robot:[250,**399**]	Y	Robot:[250,**467**]
5	N	N	Y	Robot:[**3300**,3300]	Y	Robot:[**550**,3300]	Y	Robot:[**550**,3300]
6	N	N	Y	Ram:[1714,**1714**]	Y	Ram:[1714,**2000**]	Y	Ram:[1714,**2000**]
7	N	N	Y	CNC1:[**3273**,6000] CNC2:[1714,**2667**]	Y	CNC1:[**2571**,6000] CNC2:[1714,**4000**]	Y	CNC1:[**2571**,6000] CNC2:[1714,**4000**]
8	N	N	Y	Ram:[1714,**1714**]	Y	Ram:[1714,**2000**]	Y	Ram:[1714,**2000**]
9	N	N	Y	Ram:[**2667**,6000]	Y	Ram:[**2400**,6000]	Y	Ram:[**2000**,6000]
10	N	N	Y	Ram:[**2667**,6000]	Y	Ram:[**2000**,6000]	Y	Ram:[**2000**,6000]

TABLE IV: Detection Results

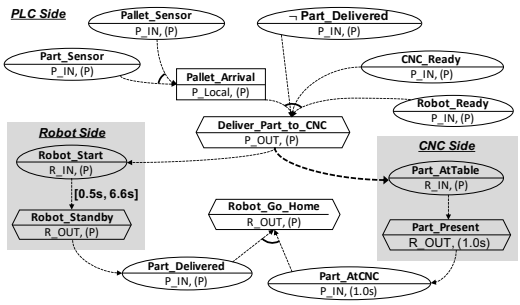


Fig. 7: A TECG of Case #2 (Robot in Danger Zone)

CNCs into multiple serial cells where up to 6 parallel machines work in the same cell on the same workloads. Thus, the amount of manufacturing steps and data communication in such a factory is comparable to that of ours. We admit that once a manufacturing system is scaled up, more computation power will be required to conduct our analysis and data mining. To address this challenge, one possible solution is to take advantage of the inherent parallelism to scale the computation. Due to the hierarchical architecture of factory floors, it is possible to divide an entire plant into multiple relatively independent groups, each of which can be analyzed individually. The summarized results of individual groups can be combined to carry out an analysis of the entire factory.

Specific Challenges to PLC Code Analysis. When compared to analyzing programs in other domains (e.g., Android apps, web programs), the analysis of PLC code is inherently unique due to three reasons. (a) PLC code controls multiple types of customized hardware constrained by unique physical limits. (b) PLC software follows a unique programming paradigm due to the introduction of PLC scan cycles. (c) Most importantly, PLC events are highly time-sensitive, due to the physical nature of machines. Such time sensitivity is the exact cause of certain safety problems discovered in our work.

IX. Related Work

Safety Verification of PLC Code. Many prior efforts [24], [28], [30], [31], [42], [44], [57], [58], [61], [63], [65] have been made to statically verify logic code using model checkers [15], [21]. Further efforts have also been made to conduct runtime verification in an online [39], [45] or offline manner [35], [62]. More recently, symbolic execution [43], [54] has been enabled on PLC code. While TSV [54] conducted static symbolic execution on its temporal execution graphs, SymPLC [43] leveraged OpenPLC [16] framework and Cloud9 engine [4] to conduct dynamic analysis. In contrast, VetPLC aims to verify real-world PLC code, which is driven by events.

Mining Temporal Invariants. Synoptic [29] and Perfume [60] extracted temporal invariants from conventional system logs via data mining. Different from OS events, ICS events are created by distributed sources on the factory floor and are difficult to obtain. Recently, ARTINALI [25] mined temporal properties from smart meters and medical devices to enable intrusion detection. To detect anomalies in ICS, Chen et al. [33] managed to learn invariants from data traces of a water purification testbed. As a comparison, VetPLC also mines ICS invariants but addresses a different problem.

Exercising Event-Driven Programs. Anand et al. [27] proposed to generate GUI event sequences based upon concolic testing. Mirzaei et al. [55] correlated events with their handlers for generating Android-specific drivers. AppIntent [67] relied on Android lifecycle model to produce event-space constraint graphs. Jensen et al. [46] built event sequences based upon concolic execution and Android GUI model. kudzu [66] developed a GUI explorer that randomly searches Web event space. SymJS [51] discovered Web event sequences via a feedback directed exploration and dynamic taint analysis. SymRT [52] performed timing analysis for real-time Java systems based upon symbolic execution and model checking. Lee et al. [48] proposed to create test sequences from Modechart specifications. In contrast, VetPLC can automatically discover both event ordering and timing without predefined specifications.

Event Causality. Orpheus [34] modeled program behaviors based upon CPS events, and applied these models to anomaly detection. Zhang et al. [68] detected malware via the inference of triggering relations between events in network data. Compared to the prior work which studied qualitative event causalities, VetPLC takes a step further and quantitatively recovers event timings that are critical for PLC code analysis.

X. Conclusion

We propose VetPLC, a novel approach to automatically produce timed event sequences for PLC code vetting. The evaluation of our prototype on two real-life ICS testbeds shows that VetPLC can effectively generate event sequences which automatically lead to hidden safety violations.

Acknowledgment

We would like to thank anonymous reviewers and our shepherd, Prof. Daphne Yao, for their feedback in finalizing this paper. This research was supported in part by NSF Grant CNS-1544613, CNS-1544901, CNS-1544678 and CNS-1718952. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] "ABB RAPID Veteran, a few question about FANUC KAREL," https://www.robot-forum.com/robotforum/fanuc-robot-forum/abb-rapid-veteran-a-few-question-about-fanuc-karel/.

[2] "Antlr," http://www.antlr.org/.

[3] "Clang: a C language family frontend for LLVM," https://clang.llvm.org/.

[4] "Cloud9 - Automated Software Testing at Scale," http://cloud9.epfl.ch/.

[5] "Conveyor Belts Optimisation," https://www.standard-industrie.com/en/wp-content/themes/standardindustrie/img/CONVEYOR_BELT_OPTIMISATION.pdf.

[6] "Conveyors and Falling Item Prevention," http://www.cisco-eagle.com/blog/2015/08/20/conveyors-and-falling-item-prevention/.

[7] "Cooperation and Control: A Systems Perspective," https://me.engin.umich.edu/news-events/news/cooperation-and-control-systems-perspective.

[8] "Ethernet/ip," https://en.wikipedia.org/wiki/EtherNet/IP.

[9] "Foundations For Conveyor Safety Book," http://martinengineerings3.s3.amazonaws.com/www.martin-eng.de/download/FoundationsForConveyorSafetyBook.pdf.

[10] "IEC 61131-3," https://en.wikipedia.org/wiki/IEC_61131-3.

[11] "Industrial Control Systems Killed Once And Will Again, Experts Warn," https://www.wired.com/2008/04/industrial-cont/.

[12] "Industry 4.0," https://en.wikipedia.org/wiki/Industry_4.0.

[13] "MATIEC - IEC 61131-3 compiler," https://bitbucket.org/mjsousa/matiec.

[14] "MATIEC examples," https://github.com/Felipeasg/matiec_examples.

[15] "NuSMV: a new symbolic model checker," http://nusmv.fbk.eu/.

[16] "OpenPLC Project," http://www.openplcproject.com/.

[17] "PLC Manufacturer Rankings," http://automationprimer.com/2013/10/06/plc-manufacturer-rankings/.

[18] "Programmable Logic Controller," https://en.wikipedia.org/wiki/Programmable_logic_controller.

[19] "Robot kills worker at Volkswagen plant in Germany," https://www.theguardian.com/world/2015/jul/02/robot-kills-worker-at-volkswagen-plant-in-germany.

[20] "Structured Text Tutorial to Expand Your PLC Programming Skills," http://www.plcacademy.com/structured-text-tutorial/.

[21] "UPPAAL Home," http://www.uppaal.org/.

[22] "ANSI/RIA R15. 06: 2012 Safety Requirements for Industrial Robots and Robot Systems," *Ann Arbor: Robotic Industries Association*, 2012.

[23] M. Abrams and J. Weiss, "Malicious Control System Cyber Security Attack Case Study – Maroochy Water Services, Australia," https://www.mitre.org/sites/default/files/pdf/08_1145.pdf.

[24] A. Aiken, M. Fähndrich, and Z. Su, "Detecting Races in Relay Ladder Logic Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[25] M. R. Aliabadi, A. A. Kamath, J. Gascon-Samson, and K. Pattabiraman, "ARTINALI: Dynamic Invariant Detection for Cyber-physical System Security," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, Sep 2017.

[26] R. Alur and T. A. Henzinger, "A Really Temporal Logic," *J. ACM*, vol. 41, no. 1, Jan. 1994.

[27] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, 2012.

[28] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Regression Verification for Programmable Logic Controller Software," in *Formal Methods and Software Engineering*, 2015.

[29] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, Sep 2011.

[30] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: A Verification Platform for Programmable Logic Controllers," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Sep 2012.

[31] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the Automatic Verification of PLC Programs Written in Instruction List," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Feb 2000.

[32] M. Chai and B.-H. Schlingloff, "A Rewriting based Monitoring Algorithm for TPTL," vol. 1032, pp. 61–72, Jan 2013.

[33] Y. Chen, C. M. Poskitt, and J. Sun, "Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System," in *2018 IEEE Symposium on Security and Privacy (Oakland'18)*, May 2018.

[34] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*, Dec 2017.

[35] J. Dzinic and C. Yao, "Simulation-based Verification of PLC Programs Master of Science Thesis in Production Engineering," Master's thesis, Chalmers University of Technology, Sweden, 2013.

[36] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier," https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[37] G. Fedorko, V. Molnar, D. Marasova, A. Grincova, M. Dovica, J. Zivcak, T. Toth, and N. Husakova, "Failure Analysis of Belt Conveyor Damage caused by the Falling Material. Part II: Application of Computer Metrotomography," *Engineering Failure Analysis*, vol. 34, pp. 431 – 442, 2013.

[38] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards Detecting Logic Bombs in Android Applications," in *2016 IEEE Symposium on Security and Privacy (Oakland)*, May 2016.

[39] L. Garcia, S. Zonouz, D. Wei, and L. P. de Aguiar, "Detecting PLC control corruption via on-device runtime verification," in *2016 Resilience Week (RWS)*, Aug 2016.

[40] A. Ginter, "The Top 20 Cyber Attacks Against Industrial Control Systems," https://ics-cert.us-cert.gov/sites/default/files/ICSJWG-Archive/QNL_DEC_17/Waterfall_top-20-attacks-article-d2%20-%20Article_S508NC.pdf.

[41] N. Govil, A. Agrawal, and N. O. Tippenhauer, "On Ladder Logic Bombs in Industrial Control Systems," in *CyberICPS/SECPRE@ESORICS*, Sep 2017.

[42] J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn, "The Safety Guaranteeing System at Station Hoorn-Kersenboogerd," in *Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, Jun 1995.

[43] S. Guo, M. Wu, and C. Wang, "Symbolic Execution of Programmable Logic Controller Code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, Sep 2017.

[44] R. Huuck, "Semantics and Analysis of Instruction List Programs," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 3–18, 2005.

[45] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-Monitoring for Industrial Control Systems," *Electronics*, vol. 4, no. 4, pp. 995–1017, dec 2015.

[46] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated Testing with Targeted Event Sequence Generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, Jul 2013.

[47] I. Kovalenko, M. Saez, K. Barton, and D. Tilbury, "SMART: A System-Level Manufacturing and Automation Research Testbed," *Smart and Sustainable Manufacturing Systems*, vol. 1, no. 1, pp. 232–261, 2017.

[48] N. H. Lee and S. D. Cha, "Generating Test Sequences Using Symbolic Execution for Event-Driven Real-Time Systems," *Microprocessors and Microsystems*, vol. 27, pp. 523–531, 2003.

[49] R. M. Lee, M. J. Assante, and T. Conway, "German Steel Mill Cyber Attack," https://ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks_Facility.pdf.

[50] R. Lee, M. Assante, and T. Conway, "Analysis of the Cyber Attack on the Ukrainian Power Grid," https://www.nerc.com/pa/CI/ESISAC/Documents/E-ISAC_SANS_Ukraine_DUC_18Mar2016.pdf.

[51] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, Nov 2014.

[52] K. S. Luckow, C. S. Păsăreanu, and B. Thomsen, "Symbolic Execution and Timed Automata Model Checking for Timing Analysis of Java Real-Time Systems," *EURASIP Journal on Embedded Systems*, vol. 2015, no. 1, Sep 2015.

[53] A. Martelli and U. Montanari, "Additive AND/OR Graphs," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, Aug 1973.
[54] S. McLaughlin, S. Zonouz, D. Pohly, and P. McDaniel, "A Trusted Safety Verifier for Process Controller Code," in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*, Feb 2014.
[55] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android Apps Through Symbolic Execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012.
[56] A. Montaqim, "Top 14 industrial robot companies and how many robots they have around the world," https://roboticsandautomationnews.com/2015/07/21/top-8-industrial-robot-companies-and-how-many-robots-they-have-around-the-world/812/.
[57] J. Nellen, E. Ábrahám, and B. Wolters, "A CEGAR Tool for the Reachability Analysis of PLC-Controlled Plants Using Hybrid Automata," in *Formalisms for Reuse and Systems Integration*, 2015.
[58] J. Nellen, K. Driessen, M. Neuhäuβer, E. Ábrahám, and B. Wolters, "Two CEGAR-based Approaches for the Safety Verification of PLC-controlled Plants," *Information Systems Frontiers*, vol. 18, no. 5, pp. 927–952, Oct. 2016.
[59] Occupational Safety and Health Administration, "OSHA Instruction PUB 8-1.3 SEP 21, 1987 Office of Science and Technology Assessment," https://www.osha.gov/enforcement/directives/std-01-12-002.
[60] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral Resource-aware Model Inference," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, Sep 2014.
[61] S. Ould Biha, "A Formal Semantics of PLC Programs in Coq," in *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference (COMPSAC'11)*, Jul 2011.
[62] S. C. Park, C. M. Park, G.-N. Wang, J. Kwak, and S. Yeo, "PLCStudio: Simulation based PLC code verification," *2008 Winter Simulation Conference*, pp. 222–228, 2008.
[63] T. Park and P. I. Barton, "Formal Verification of Sequence Controllers," *Computers & Chemical Engineering*, vol. 23, no. 11, pp. 1783–1793, 2000.
[64] B. Perelman, "The Top 3 Threats to Industrial Control Systems," https://www.securityweek.com/top-3-threats-industrial-control-systems.
[65] J.-M. Roussel and B. Denis, "Safety Properties Verification of Ladder Diagram Programs," *Journal Européen des Systèmes Automatisés (JESA)*, vol. 36, no. 7, pp. pp. 905–917, Jun. 2002.
[66] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland'10)*, May 2010.
[67] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS'13)*, Nov 2013.
[68] H. Zhang, D. D. Yao, N. Ramakrishnan, and Z. Zhang, "Causality Reasoning About Network Events for Detecting Stealthy Malware Activities," *Computers and Security*, vol. 58, no. C, May 2016.

APPENDIX

A. Teach Pendant Code of FANUC Robot

Figure 8 presents the robot code implemented using teach pendant language. This program is triggered by a PLC event and can pass a part from CNC machine to conveyor.

```
1 !Function only when receiving the signal
2 IF DI[0:Deliver_Part@PLC]=OFF, JMP LBL[3]
3 DO[6:Pickup_from_CNC1]=ON
4 DO[2:Part_AtConveyor@PLC]=OFF
5 CALL GO_HOME_AND_GET_VACUUM_GRIPPER
6 !Move to CNC1
7 J P[10:ROTARM] 80% FINE
8 L P[4:ROTARM2] 250mm/sec FINE
9 ...
10 !Pick up a part from CNC1
11 L P[9:CNCSIDE] 100mm/sec FINE
12 ...
13 LBL[1]
14 IF DI[7:Pickup_Confirmation]=ON, JMP LBL[2]
15 JMP LBL[1]
16 LBL[2]
17 WAIT .10(sec)
18 !Deposit part on conveyor
19 L P[10:ROTARM] 550mm/sec FINE
20 ...
21 !Notify that part was dropped on conveyor
22 DO[2:Part_AtConveyor@PLC]=ON
23 WAIT .50(sec)
24 DO[2:Part_AtConveyor@PLC]=OFF
25 CALL RETURN_VACCUM_GRIPPER_AND_GO_HOME
26 DO[6:Pickup_from_CNC1]=OFF
27 LBL[3]
```

Fig. 8: Robot Teach Pendant Code for Delivering Parts

B. Implementation of Static Analysis

We have implemented our *static analyses* in 7K lines of C++ code and 5K lines of Java code. Particularly, we convert PLC ST code into C programs via MATIEC [13] compiler, and then leverage Clang [3] to enable our analyses. To analyze teach pendant programs in robot, we build a specific parser using Antlr [2] and then perform control flow analysis on top of the generated AST.

Note that the conversion from PLC to C code, using MATIEC, follows a standardized (IEC 61131-3) mechanism. We admit that some semantics, such as counters, timers, etc. may not be very precisely translated to C code especially because of the implicit effects caused by PLCs' scan cycles. Furthermore, different vendors may introduce unique features, besides standard ones, that cannot be converted using existing tools. To address these limitations, an alternative option is to directly conduct analysis on native PLC code. We intend to work on this as part of future work. However, we argue that our graph construction methods are orthogonal to the underlying program analysis. In fact, other (potentially advanced) analysis techniques can be used to achieve our goal.

C. Algorithm to Infer Events From Data Traces.

Algorithm 3 depicts our algorithm to infer discrete events from continuous data traces collected from physical ICS testbeds.

D. Example of Event Sequence & Implementation

Motivating Example Figure 9 depicts how we apply a generated event sequence to exercising PLC code of the motivating example. In this chart, the x-axis represents time (in seconds), which is ranging from Begin-of-Test (BOT) to End-of-Test (EOT), and the y-axis denotes the list of events. The effective duration of each event is illustrated as a thick horizontal line, which begins with an empty circle and ends with a filled circle or a cross. The filled circle means the event is terminated by its sender, and the cross indicates it is disabled due to PLC logic. The dotted part on a thick line represents the possible range of starting point of an event. For instance, the

Algorithm 3 Event Inference

```
1:  procedure InferEvents(𝔻𝕋)
2:      𝔼𝕋 ← ∅
3:      for ∀DT_i ∈ 𝔻𝕋 do
4:          ET_i ← ∅
5:          {DT_i^{v0}, DT_i^{v1}, ..., DT_i^{vk}} ← DivideByVar(DT_i)
6:          for ∀DT_i^{vp} ∈ {DT_i^{v0}, DT_i^{v1}, ..., DT_i^{vk}} do
7:              {d'_0, d'_1, ...d'_m} ← DT_i^{vp}
8:              l ← 0
9:              for l < m do
10:                 if d'_l ≠ d'_{l+1} then
11:                     e ← (state_{d'_{l+1}}, time_{d'_{l+1}})
12:                     ET_i ← ET_i ∪ e
13:                 end if
14:                 l ← l + 1
15:             end for
16:         end for
17:         SortByTime(ET_i)
18:         𝔼𝕋 ← 𝔼𝕋 ∪ ET_i
19:     end for
20: end procedure
```

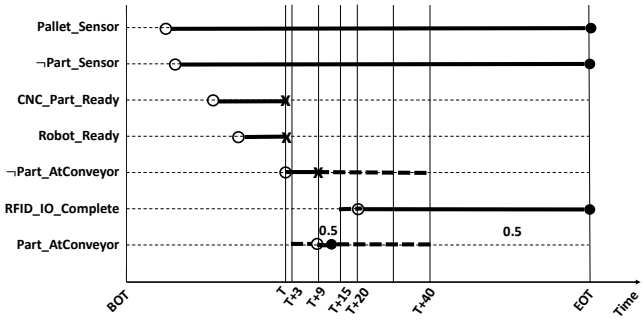


Fig. 9: Generating Event Sequence for Motivating Example

starting time of event `Part_AtConveyor` ranges from $T+3$ to $T+39.4$ seconds due to the variation of robot delivery time, where T is the time to signal `Deliver_Part`. Similarly, the beginning of `RFID_IO_Complete` is from $T+19$ to $T+20$.

This chart shows one permutation of the 7 input events. Since the five events on the top do not bear any temporal dependencies, they can be arranged in any orders, one of which is depicted here. These 5 events are "Permanent" ones and are always enabled until programmatically disabled (e.g., `CNC_Part_Ready`, `Robot_Ready` and `¬ Part_AtConveyor`). Then, the starting timestamps of `RFID_IO_Complete` and `Part_AtConveyor` are relative to the timestamp T at which all these five events have been triggered. We discretize time with ρ being 5 and, in this permutation, we choose to include one discrete version for both of them, $\texttt{RFID\_IO\_Complete}_{T+20}$ and $\texttt{Part\_AtConveyor}_{T+10}$. While $\texttt{RFID\_IO\_Complete}_{T+20}$ is a long-lasting event, $\texttt{Part\_AtConveyor}_{T+10}$ becomes inactive at $(T+10)+0.5$. In consequence, this sequence will trigger the aforementioned error because $\texttt{Part\_AtConveyor}_{T+10}$ is turned off prematurely.

Implementation. We simulate each sequence of events through refreshing values of PLC variables when their corresponding events occur. We then persist the result values into a file, which is accessed by PLC code at the beginning and end of every scan cycle. This is to mimic the input and output phases of a PLC cycle. To reflect the potential event termination originated from PLC logic, we compute a conjunction between each generated event and its current state in PLC, and use the result as the new input. For events with certain durations, we set up timers to control their active periods.

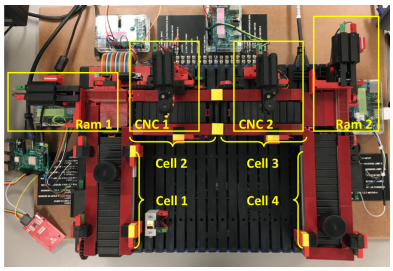


Fig. 10: Fischertechnik Testbed for Manufacturing System

E. Details of TPTL Grammar

The grammar of TPTL is built from proposition symbols and timing constraints by Boolean connective, temporal operators, and freeze quantifiers.

The *Timing Constraints* of TPTL are of the form $\pi_1 \leq \pi_2$ and $\pi_1 \equiv_d \pi_2$ (time π_1 is congruent to time π_2 modulo the constant d). The abbreviations x (for x + 0), = , $<$, $>$, $\geq$, true, $\neg$, $\wedge$, and $\vee$ are defined as usual.

The *Temporal Operators* can be either 1) *next* formula $\bigcirc p$ that asserts about a timed state sequence that the second state in the sequence satisfies the proposition p, or 2) *until* formula $p_1 U p_2$ that asserts about a timed state sequence that there is a state satisfying the proposition p_2, and all states before this p_2-state satisfy the proposition p_1. Additional temporal operators are defined as usual. In particular, the *eventually* operator $\Diamond\phi$ stands for $trueU\phi$, and the *always* operator $\Box\phi$ stands for $\neg\Diamond\neg\phi$.

The *Freeze Quantifier* can be associated to a variable x as "x." and it freezes x to the time of the local temporal context.

F. Fischertechnik Testbed

This testbed is divided into four cells (Cell 1 to Cell 4), each of which is equipped with a conveyor belt and one or two IR sensors that detect the presence of parts and is controlled by a PLC. The testbed contains two CNC machines (CNC 1 and CNC 2) located in Cell 2 and Cell 3 respectively. Two rams (Ram 1 and Ram 2) are deployed to move parts from Cell 1 to Cell 2 and from Cell 3 to Cell 4 respectively. These CNC machines and rams are also controlled by separate PLCs. In this testbed, a PLC is emulated by a Raspberry Pi board running an OpenPLC server to execute PLC code. All Raspberry Pi boards are connected together via Ethernet and linked via Modbus.

The system starts when a part enters the manufacturing line from Cell 1 and is passed to Cell 2 by Ram 1 for the operation processed by CNC 1. The part is then moved to Cell 3 for the operation processed by CNC 2. When both CNC operations

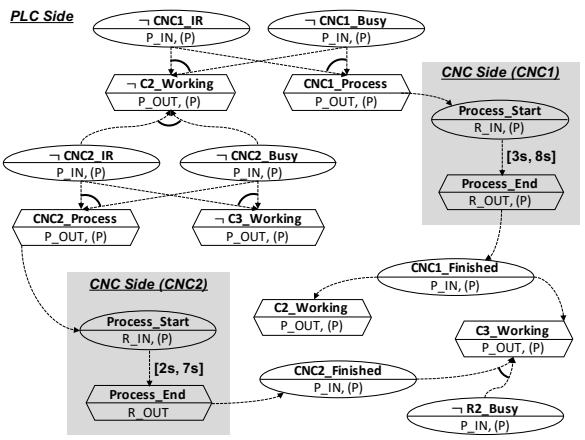


Fig. 11: A TECG of Case #7 (CNC-Part Collision)

are complete, the part is transferred to the conveyor in Cell 4 by Ram 2 and leaves the testbed.

It is possible to place multiple parts on the testbed at the same time and process the parts sequentially. However, due to physical limitations in the testbed (e.g., limited length for the conveyor belt, long operation time for the rams and CNC machines), restrictions should be taken into account when developing the control logic.

G. Case Study on Scenario #7 CNC-Part Collision

Description. This case focuses on the section where a part is processed by CNC 1 and to be transferred to CNC 2. Since the testbed has a linear setup, the design and deployment of the CNC machines are based upon an assumption: *when a CNC finishes an operation and is ready to discharge a part, its successive CNC should also be ready to receive the part* – this avoids a downgrade in system throughput due to congestion in the linear model. That is, in this case, CNC 2 is expected to be ready (i.e., the preceding part has been discharged from CNC 2) when CNC 1 finishes a process and discharges a part. In a normal manufacturing run, CNC 2 sends a signal to PLC when a part is processed. PLC then activates Conveyor 3 to transfer the part from CNC 2 to the next cell (Ram 2). Similarly, when a part is processed by CNC 1, Conveyor 2 and 3 are activated by PLC to transfer the part from CNC 1 to CNC 2.

A potential issue may occur in this linear setup when the aforementioned assumption no longer holds due to changes in time correlation between CNC machines. This could happen either because of a worn-out component in a CNC that leads to a longer CNC cycle time or a careless change in manufacturing plan (e.g., an operator speeds up the conveyor with a desire for higher production performance).

Safety Vetting. Using the proposed analysis method, we first construct the TECG (as shown in Figure 11) by analyzing the PLC and CNC code. In this case, the correlation between the two CNC machines and PLC can be revealed in this step. From this TECG, we can determine that the event CNC2_Process is followed by the event CNC2_Finished and the event CNC1_Process is followed by the event CNC1_Finished. These event dependencies discovered from the inter-device communication help reduce the number of possible permutations from 13700 to 898 (without taking time into account). Then, we proceed to the temporal property mining process that produces time correlation and temporal invariants for the events. In this case, process times, $T_{CNC1\_Process}$ and $T_{CNC2\_Process}$, from both CNC machines are obtained, which are associated to the time durations for Process_Start $\rightarrow$ Process_End of both CNCs. With these time invariants being considered, the number of permutations becomes 6442, 24358 and 79818 for VETPLC-TSEQS-2, VETPLC-TSEQS-5 and VETPLC-TSEQS-10, respectively.

In this case, CNC 1 process time, $T_{CNC1\_Process}$, ranges from 3 to 8 seconds and CNC 2 process time, $T_{CNC2\_Process}$, ranges from 2 to 7 seconds. As mentioned above, anomalies may occur when either CNC 2 takes longer to finish its task or CNC 1 discharges a part earlier. Under either circumstance, it is possible that the part discharged from CNC 1 arrives in CNC 2 before the precedent part originally in CNC 2 fully leaves the cell. As a result, the successive part may collide with the preceding part as well as CNC 2 and cause safety issues. This violates the safety specification, $\square$(CNC_Busy $\rightarrow$ $\neg\Diamond$Part_Arrival), that indicates that a part must not arrive at a CNC when it is in a busy state. Through VETPLC-TSEQS test processes, the system determines that this violation may occur when CNC 1 is running at a speed from 3273 rpm to 6000 rpm and CNC 2 is running at a speed from 1714 rpm to 2667 rpm with VETPLC-TSEQS-2. The same violation can also be captured using VETPLC-TSEQS-5 and VETPLC-TSEQS-10 with higher precision with respect to the error-triggering speed ranges (see Table IV for details).