# Automated Runtime Mitigation for Misconfiguration Vulnerabilities in Industrial Control Systems

Qingzhao Zhang
University of Michigan
United States
qzzhang@umich.edu

Xiao Zhu
University of Michigan
United States
shawnzhu@umich.edu

Mu Zhang
University of Utah
United States
muzhang@cs.utah.edu

Z. Morley Mao
University of Michigan
United States
zmao@umich.edu

## ABSTRACT

Cyber-physical industrial control systems (ICS) commonly implement configuration parameters that can be remotely tuned by human-machine interfaces (HMI) at runtime. These parameters directly control the behaviors of ICSs thus they can be exploited by attackers to compromise the safety of ICSs, proved by real-world attacks worldwide. However, existing anomaly detection methods, which mostly focus on the programmable logic controller (PLC) programs or sensor signals, lack a comprehensive analysis of configuration's impact on the entire system and thus cannot effectively detect improper parameters. A tool that automatically analyzes complicated control logic to determine the safety of configuration is absent. To fill this gap, we design SmtConf, a verification-based framework for detecting and mitigating improper parameters in ICSs at runtime. To understand the impact of configuration parameters on complicated control logic, we design a symbolic formal model representing behaviors of the ICS under any possible configuration parameters. Based on the model, SmtConf works as a monitoring system that detects safety violations in real-time when the improper configuration is injected. To further assist developers to determine the safe configuration, SmtConf recommends safe configuration parameters by solving an optimization problem. In 18 test cases collected from two production-level ICS testbeds, SmtConf detects all true violations caused by improper parameters in 0.41 seconds and correctly repairs the ICS with recommended safe parameters in 0.45 seconds.

## CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Software and its engineering** → *Formal software verification.*

## KEYWORDS

Industrial Control System; Formal verification

**ACM Reference Format:**
Qingzhao Zhang, Xiao Zhu, Mu Zhang, and Z. Morley Mao. 2022. Automated Runtime Mitigation for Misconfiguration Vulnerabilities in Industrial Control Systems. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3545948.3545954


*RAID 2022, October 26–28, 2022, Limassol, Cyprus*

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545954

## 1 INTRODUCTION

Safety hazards in industrial control systems (ICS) have become a serious problem and thus attracted significant attention. Reports have shown that many fatal incidents were actually caused by intentional attacks [1–3, 19, 35, 36, 47]. Many efforts [11, 13, 18, 23, 25, 29, 40, 42, 43, 56] have been therefore made to automatically detect malicious attacks and prevent safety problems in ICSs.

Nevertheless, prior work largely depends on a very strong assumption that adversaries must be able to either directly modify controller code (and restart an entire control system) [4, 25, 40, 54] or have full control over the industrial network and thus can inject arbitrary control data [37]. However, a recent incident in the Florida water treatment plant [47] has demonstrated that attacking control systems can be easily enabled via altering critical control parameters using legitimate yet exposed human-machine interfaces (HMI). Compared to code modification attacks, modifying configuration can be achieved at runtime and does not require highly suspicious system reboots. In contrast to injecting fake sensor data into the air-gapped industrial network, manipulating configurations at common interfaces is much easier and thus significantly decreases the bar for causing physical damage at the same levels.

In fact, this new class of simple but stealthy attacks may render existing defense techniques ineffective. Targeting code modification and sensor data injection, state-of-the-art attack detectors rely on discovering inconsistencies among various data points in the industrial system. The consistency rules can be defined by the control invariants implemented in PLC code [4, 15, 54] or intrinsic dependencies among sensor measurements according to physical laws [7, 53]. However, the above inconsistencies do not appear in configuration manipulation attacks which change neither the code logic of PLCs nor the values of sensor measurements. Instead, improper configuration parameters exploit existing design defects in the ICS as the developers cannot perfectly restrict the configuration parameters to exclude all potential safety risks. To detect misconfiguration, we must completely enumerate behaviors of the ICS by considering configuration parameters along with PLC control logic and physical laws as an entirety, instead of focusing on only PLCs or sensors. Only by understanding the interaction between ICS configuration and all other system variables, we can quantitatively and precisely measure the impact of the ICS configuration parameters to evaluate whether a configuration choice is safe.

To close the gap, we propose to detect improper configuration parameters at runtime using formal verification. For verification, it is a common approach to construct a formal model to represent the

behaviors of the system to be analyzed. The design of the formal model is crucial as it decides the capability of verification applications we can support. Previous modeling of ICSs [22, 25, 26, 40, 56] do not consider dynamic system parameters and their verification verifies one specific system configuration in each run. As a result, existing modeling approaches fail to efficiently analyze the safety of the infinite possible choices of configuration parameters. To address the problem, we design a novel formal model by introducing symbolic configuration parameters which interact with other system variables from PLCs, machines, or sensors. We then design model checking methods to analyze the effect of configuration changes in the complicated ICS environment by considering program logic, physical laws, and temporal dependencies altogether. Based on the formal method, we naturally build various applications including fast detection of configuration manipulation and recommendation of optimal safe parameters.

We design and implement a framework SmtConf for mitigating improper configuration in ICSs, considering both malicious attackers and careless operators. SmtConf first leverages program analysis on the code of programmable logic controllers (PLC) and data mining on real data traces to semi-automatically construct a model of the ICS offline, named Symbolic Event Sequence Graph (SESG). SESG supports symbolization on certain configuration parameters and leverages satisfiability modulo theories (SMT) to formally reason the reachability of system states. By analyzing the SESG, SmtConf generates a safety constraint offline, which indicates the safe ranges of configuration parameters that are violation-free against a set of safety specifications provided by system developers. We focus on safety specifications about timing-based hazards, *i.e.,* wrong parameters cause wrong temporal orders of events, which is the major type of safety hazard in ICSs according to previous studies [56]. The online mitigation algorithm is based on the ICS model and is deployed on a bump-in-the-wire isolated device between controllers and HMIs. While the ICS is running, SmtConf intercepts updates of configuration parameters and checks whether the new configurations will violate the safety constraint by solving a constraint satisfaction problem. If a violation is detected, SmtConf blocks the update. If asked by ICS operators, SmtConf also recommends safe and optimal configuration values through mathematical optimization, as guidance for ICS operators to correctly configure the ICS. Under the assumption of a sound ICS model, the violation detection has zero false negatives and the recommended parameters are violation-free. As an automatic tool, SmtConf can gain time for human experts to thoroughly investigate detected problems in order to create a permanent solution.

We address several challenges to make our framework deployable and useful in real-world ICSs. First, the online mitigation must have low latency to deliver benign configuration updates in time. To address the problem, SmtConf conducts a thorough analysis of the ICS model offline to generate the safety constraint. As a result, the online processes just solve lightweight mathematical problems on the constraint without processing complicated system models. Second, the recommendation of configurations should not only be violation-free but also satisfy the developer's intention. For example, developers may need safe configurations which can maximize system production. To enable this functionality, we transform the recommendation problem into a mathematical optimization problem whose objective function can be customized.

To demonstrate the effectiveness of SmtConf, we tested it on two ICS testbeds: SMART [34] and Fischertechnik [20]. On each testbed, SmtConf spends around 20 minutes preparing the system model and safety constraints. We then collected 18 test cases (*i.e.,* ICS scenarios) among which 15 cases have improper configurations of the testbeds. For the 3 test cases with safe system configurations, SmtConf verifies its safety and assures that they are violation-free under the assumption of sound system modeling. Our simulation of the testbeds does not observe any missed violation (*i.e.,* false negative) in the 3 cases. For the 15 test cases with violations, SmtConf successfully detects the problem in 0.4 seconds for each case and we confirm that all detected violations are true positives according to manual validation as well as simulation. SmtConf also provides the recommendation of safe and optimized configurations in 0.45 seconds for each violation case. Compared to previous ICS safety verification work VetPLC, SmtConf's ICS model analyzes 216% more possible execution traces and discovered 2 more hazard cases while the analysis time is shortened by 100× to 1,000×.

Our contribution is summarized as follows:

- We design and implement SmtConf, a verification-based framework that detects configuration manipulation attacks and automatically adjusts the parameter values to an optimal and safe state. The framework systematically integrates model checking, static program analysis, and data mining to achieve efficient verification.
- We design a novel modeling method for ICSs, Symbolic Event Sequence Graph (SESG), from which one can reveal potential safety hazards and calculate the safe ranges of configuration parameters.
- We test our prototype on two ICS testbeds and prove the correctness and efficiency of our approach.

## 2 BACKGROUND

**Industrial Control System** (ICS) is the computer system used to monitor and control industrial processes. Examples of ICSs include power grids, transportation, and manufacturing sectors. A typical ICS contains numerous control loops, human interfaces, as well as diagnostics or maintenance tools built on a network protocol for communications [30]. Sensors (*e.g.,* cameras), actuators (*e.g.,* robots), and controllers (*e.g.,* PLCs) work together to implement the control loops. Operators or developers use human-machine interfaces (HMIs) to monitor the system status and to configure parameters in the controller or other devices. Especially, PLC is the core control unit handling complex control logic. A typical PLC runs programs that embed control logic to generate control commands based on sensor measurements.

**Satisfiability Modulo Theories** (SMT), as an extended form of Boolean satisfiability (SAT), represents a constraint solving problem expressed in classical first-order logic [8]. SMT can represent theories of integers, real numbers, and various data structures including bit-vectors and arrays. Then SMT solvers (*e.g.,* Z3 [17]) can automatically solve the SMT problem defined by the above theories. In research of computer science, SMT is commonly used as a tool of formal verification and its applications include bug or vulnerability

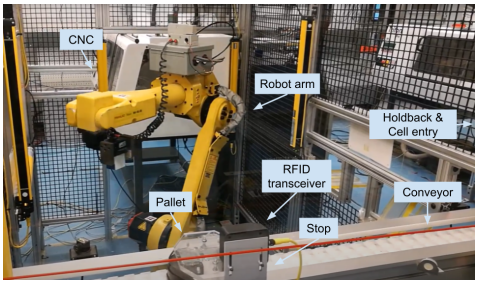



**Figure 1: Devices in the SMART cell. A robot arm is delivering a part from an on-conveyor pallet to a CNC.**

detection [28, 52], test case generation [12], correctness verification of system software [44, 50, 57], *etc.* Especially, in order to use SMT for vulnerability detection, one can define the existence of potential vulnerabilities using the language of SMT, use an SMT solver to solve the constraint satisfaction problem, and discover tricky vulnerabilities more efficiently than black-box methods such as fuzzing and simulation.

## 3 MOTIVATION

In this section, we use one real ICS incident to motivate our study on ICS runtime mitigation of misconfiguration vulnerabilities.

### 3.1 Motivating Example

We motivate the problem of ICS misconfiguration using SMART [34], which is a manufacturing sector testbed built by Rockwell Automation (a leading provider of industrial automation in North America) and equipped with Allen Bradley PLCs and FANUC robots. The whole system produces toy cars and our motivating example focuses on one cell which processes one part of the final production.

**Overall workflow.** Figure 1 shows the devices in the SMART cell including two gates (*i.e.,* holdback and stop), one pick-and-place robot arm, one computer numerical control (CNC) machine, and sensors including one radio-frequency identification (RFID) transceiver. A PLC has connections with all the above devices. The cell executes the following steps to process one part: (1) a pallet holding a part moves in along the conveyor and passes the holdback gate; (2) the pallet then is blocked by the stop and the RFID transceiver updates the state of the part's RFID; (3) the robot arm picks up the part from the pallet and delivers it to an unoccupied CNC; (4) the CNC processes the part; (5) the robot arm delivers the finished part from the CNC back to the pallet; (6) the RFID is updated again when the part enters the detection range of the RFID transceiver; (7) finally, the stop releases the pallet.

**Configurable timing parameters.** Note that the time it takes for a CNC to process one part is configurable during runtime through HMI. Usually, developers prefer higher motor frequency hence shorter CNC processing time to improve the overall production.

**Control logic.** Listing 1 shows the PLC's control logic (pseudo-code) coordinating the devices in the cell. The original code is written in Ladder Logic language but in Listing 1 we use pseudo-code for demonstration. The code handles an RFID update process which is triggered when a part enters the detection range of the RFID transceiver (line 8). The RFID update is implemented in a state machine whose state is the integer variable `UpdateStep_RFID`. To make sure the RFID update works correctly, every time a part enters the detection range, `UpdateStep_RFID` should be reset to zero. To implement this reset behavior, timer `UpdateFinishedDelay` records the accumulated time since no part is detected (line 15, `TON` means "Timer ON"). When the accumulated time reaches a threshold (*i.e.,* 4 seconds), `UpdateStep_RFID` is reset (line 18).

The design of `UpdateFinishedDelay` timer intends to avoid glitches such as detection failures of the RFID transceiver. Even if the part is out of detection for a short time, the RFID update process is not intercepted. This case demonstrates a common and necessary usage of timers in ICS for fault tolerance.

```
1  if DeBounce2.DN and RFID_AtStop and not Robot_Busy and
2    not CNC_Busy and UpdateStep_RFID == 15:
3      Conveyor2CNC_Delivery_Start = true
4
5  if DeBounce2.DN and not RFID_AtStop and CNC_Done and not Robot_Busy:
6      CNC2Conveyor_Delivery_Start = true
7
8  if not RFID1_Busy and UpdateStep_RFID == 0
9                    and Pallet_AtStop and RFID_AtStop:
10     UpdateStep_RFID = 1
11
12 ... # UpdateStep 1...14 performing RFID read-write-read
13
14 if UpdateStep_RFID == 15 and not RFID_AtStop:
15     TON(UpdateFinishedDelay)
16
17 if UpdateStep_RFID == 15 and UpdateFinishedDelay.DN:
18     UpdateStep_RFID = 0
```

**Listing 1: Part of control logic of SMART cell PLC.**

**Safety violations.** One safety specification is that the RFID update process should be reset when every part enters the detection range of the RFID transceiver. However, the specification may be violated. In brief, the specification requires that the event "RFID_UpdateStep reset" is before "Part enters RFID range". Since the timer `UpdateFinishedDelay` has a four-second timeout, two events "Part leave RFID range" and "RFID_UpdateStep reset" have a fixed interval of 4 seconds. As a result, the total time between "Part leave RFID range" and "Part enter RFID range", which includes the time of CNC processing, must be longer than 4 seconds to satisfy the specification. When the motor frequency is 3,000 rpm, the CNC processing time is around 4 seconds and the specification is always satisfied. However, if we increase the CNC's motor frequency to shorten the CNC processing to 2.5 seconds, it is possible that the part enters the RFID detection range but the RFID reset does not happen. In this case, the timer will stop counting and never trigger the subsequent RFID update. Therefore, too high CNC motor frequency is an improper configuration we need to get rid of.

**Absence of runtime mitigation.** We observed a real situation where one operator of SMART tunes the CNC's configuration at runtime and triggered the above safety violation. However, the SMART system had no response to the improper configuration change and continued producing defective products. Malicious attackers can also leverage this vulnerable parameter to exploit the system (details in §3.2). Learned from the above accident, a mitigation mechanism is in need to detect misconfiguration problems and automatically propose safe parameters.

### 3.2 Threat Model

Given the motivation example, we define the problem of mitigating misconfiguration in ICSs and corresponding attack models.

**Timing-related safety hazards and configuration**. For ICS safety, we target temporal safety properties as previous verification-based work did [26, 40, 56]. Timing is a critical property for modeling control systems because individual operations have stringent safety requirements on speeds and cooperative industrial processes must strictly follow certain temporal orders. Failing to do so may lead to disastrous consequences such as fatal collisions.

The ICSs we target have parameters that are configurable at runtime, usually via HMIs. Since safety hazards are timing-based, we focus on the configuration parameters that affect temporal properties, *e.g.,* various timers, speed of moving devices, *etc.* Also, SmtConf assumes that the ICS can be modeled as a finite-state system. It is a common assumption used by related verification-based studies [22, 25, 40, 56].

**Configuration manipulation attack.** We focus on the attack scenario where certain parameters are improperly modified at runtime through predefined user interfaces. We consider two types of adversaries: malicious outsiders and benign insiders. The malicious outsiders are attackers who hack user interfaces through general cyberattacks to maliciously change the configuration parameters of the system. The configuration manipulation attack does not assume the attackers can inject malicious code into PLCs or spoof fake sensor signals. As an example, in the recent attack on Florida city's water supply [47], the attacker controls one worker's computer through a desktop control application (i.e., TeamViewer) and increases the amount of sodium hydroxide in the water plant by 100×. Attacks on Indian unclear plant [1] and Toyota manufacturing plant [2] are also through HMIs and malicious configuration parameters. The benign insiders are careless operators who improperly change the configuration parameters in legitimate ways, as shown in the motivating example. Without automatic safety analysis, these developers or operators have no assurance of the correctness of configuration tuning and may make mistakes.

**Out-of-scope attacks.** First, we do not defend malicious insiders who can access the defense system physically. We deploy the defense on a bump-in-the-wire isolated environment between controllers and HMIs, similar to related works [4, 40]. We also assume the communication from SmtConf has proper integrity checking and authentication. Attackers cannot compromise SmtConf's communication so that SmtConf can observe the real system status. Second, we do not defend against sensor spoofing attack [37] and control logic injection [55], which directly compromise physical devices or internal networks. These attacks require stronger attack capabilities than configuration manipulation and can be defended by a series of existing studies [7, 16, 54].

**Possible attacks on the defense system.** First, attackers may launch adaptive attacks, *i.e.,* manipulating parameters precisely to damage the system without being detected. Ideally, with a sound model, SmtConf guarantees the completeness of the detection as a verification method thus the adaptive attack is infeasible. We discuss possible inaccuracies the attackers may leverage in §8. Second, attackers may persistently change configuration values to repeatedly trigger the detection algorithm, hoping to cause Denial of Service (DoS). However, it is already a strong attack signal and thus can be easily observed and blocked by admins.

**Table 1: Existing verification-based ICS/PLC defenses.**

| Tool | Modeling | | | | | Verify | Repair |
|---|---|---|---|---|---|---|---|
| | PLC multitask | PLC Logic | Interaction of devices | Quantified timing | Dynamic parameters | | |
| SymPLC [25] | √ | √ | × | × | × | √ | × |
| TSV [40] | × | √ | × | × | × | √ | × |
| HyPLC [22] | × | √ | × | × | × | √ | × |
| UBIS [26] | × | √ | √ | √ | × | √ | × |
| VetPLC [56] | × | √ | √ | √ | × | √ | × |
| TARTAR [31] | × | × | × | √ | × | √ | √ |
| **SmtConf** | × | √ | √ | √ | **√** | √ | **√** |

## 3.3 Limitation of Existing Work

As mentioned in §1, anomaly detection methods based on inconsistencies of control data or sensor measurements [7, 16, 54] are not effective for misconfiguration. We must reveal the casual relation between configuration and system behavior to capture the vulnerabilities. To this end, formal verification is a suitable approach.

We summarize the state-of-the-art verification-based PLC/ICS defenses in Table 1. SymPLC [25] is a symbolic execution tool checking low-level PLC execution (*e.g.,* multi-task scheduling), which is not relevant to configuration parameters. TSV [40] and HyPLC [22] apply formal methods on PLC code and assume the PLC takes arbitrary inputs restricted by manually defined constraints. However, without modeling the timing in the physical environment, the PLC input constraint can hardly be identified in complicated scenarios such as our motivating example. To include the timing relation, UBIS [26] uses timed automata and VetPLC [56] samples execution traces based on a custom timed causality graph. Unfortunately, system models from neither UBIS nor VetPLC can model mutable configuration parameters thus they cannot reason the impact of those parameters on ICS tasks. Once a parameter is changed, their best solution is to redo the verification process to update the model, which introduces unaffordable latency to the violation detection. To fill the gap, SmtConf should consider all possible values of those parameters when modeling ICSs.

For model repairing, TarTar [31] can repair the bounds of clock values in timed automaton models but there is a gap between repairing abstract automaton models and recommending configuration parameters in the real-world ICSs. First, TarTar cannot repair system configuration because the timed automata model lacks symbolization on configuration parameters, as mentioned before. Also, TarTar minimizes the number of modified clock bounds but ICS developers need to customize optimization goals to achieve the trade-off between safety and production. Differently, SmtConf achieves the first practical ICS parameter recommendation by adopting symbolization, enhancing customizability, and minimizing latency.

# 4 SMTCONF OVERVIEW

We present SmtConf, a framework for mitigating misconfiguration vulnerabilities in ICSs through formal methods. SmtConf can detect malicious attacks or benign faults on system configurations and recommend safe configuration parameters.

**Architecture**. Typically, HMIs are connected to controllers via certain communication protocols [27]. SmtConf is deployed in an isolated bump-in-the-wire device between the controller and the HMI for intercepting configuration updates. For instance, we implement SmtConf on a Raspberry Pi with EtherNet communication to the PLCs and the HMIs. In large-scale supervisory control and data acquisition (SCADA) systems, user interfaces are organized

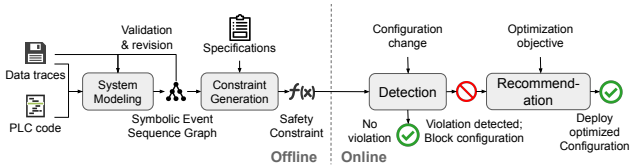



**Figure 2: Overview of SmtConf framework.**

in a hierarchy structure and SmtConf should be deployed on each lowest-level HMIs that has a direct connection with controllers.

**Components.** SmtConf has its offline module handling *system modeling* and *safety constraint generation*, as well as the online module for *detection* and *recommendation* (Figure 2).

*System modeling* (§5) is a semi-automatic process to generate an ICS system model. We apply program analysis on the PLC code and data mining on runtime data traces to extract system behaviors. We then construct a directed graph called Symbolic Event Sequence Graph (SESG) whose paths represent the possible execution of the ICS. However, the data mining is not guaranteed to be accurate thus we require developers to revise the model according to feedback from an automatic validation process.

*Constraint generation* (§6.2) accepts a set of safety specifications as input. It then analyzes the SESG to generate a constraint that indicates the safe range of parameters. The safety constraint is used by online processes.

*Detection and recommendation* (§6.3) of violations is triggered whenever configuration parameters update. It verifies whether the new configuration satisfies the safety constraint. If not, SmtConf blocks the parameter update. If requested by system operators. SmtConf can recommend the violation-free value of configuration parameters which is optimized towards a user-provided objective.

In the following sections, we introduce the detailed design of four components respectively. We use the motivating example (§3.1) to demonstrate our design.

# 5 SYSTEM MODELING

One straightforward approach to modeling an ICS is to enumerate its possible behaviors. ICS is a complicated system with heterogeneous devices and existing methods have inaccuracy problems due to incomplete modeling (§3.3). Therefore, we model the ICS as a finite state system considering various key factors including system states (§5.1), PLC execution model (§5.2), and temporal events (§5.3). We finally build a graph to represent possible behaviors of the ICS considering dynamic configuration parameters (§5.4).

## 5.1 System State

The system state consists of *state variables* extracted from the system. Different system states have different values on the state variables. The state variables have the following categories:

(1) *Signals among devices.* Complicated tasks handled by ICSs require cooperation among various devices, which is accomplished by letting devices send signals to each other via either network-based or wired connections. Typically, sensors send signals to PLCs and PLCs then interact with robots and machines. Thus the signals are important variables indicating the working status of the devices.

(2) *In-memory variables.* PLCs leverage programmable memory to handle complex dynamic control logic. During the PLC execution, a set of locally defined variables are stored in the memory and updated in each execution cycle. These variables are tangled with PLC's control logic and thus affect the system behavior.

(3) *Physical status of objects.* The sole sensor signals cannot depict the whole picture of the physical world so we add extra variables to fill the gap. One typical physical status in the motivating example is the position of pallets and parts. Though the physical parameters (*e.g.,* position) are often continuous, we can discretize them according to the discrete control logic. For instance, the pallet has states including "at stop", "passing stop", "passed stop", *etc.* Changing the pallet state means changing some sensor signals.

Next, we analyze the transition among system states, which contains two categories: PLC execution and temporal events.

## 5.2 PLC Execution Model

PLCs iterate their execution cycles with a fixed small time interval (*e.g.,* 60 ms for SMART's PLCs) and directly modify state variables. Formally, in each cycle, we model the PLC execution as a function that accepts state variables as input and produces the same set of state variables with updated values.

However, the modeling of the PLC execution is a tradeoff between precision and scalability. First, we ignore low-level features of PLCs, including the cycle time and multi-task scheduling. Modeling each PLC cycle separately is expensive and not scalable. For instance, TSV's symbolic model checking [40] costs 20 minutes on only 14 PLC cycles and the time cost increases exponentially *w.r.t.* cycle count. Instead, we regard PLC's updates of state variables as instant operations that take zero time in our SESG model (details are in §5.4). Though introducing errors on the timestamp of state variable updates, the abstraction greatly reduces the state space of our model. The error of timestamps is minor – PLC scan cycle time is less than 1% of average time intervals between system events, according to existing ICS testbeds [27]. In addition, we alleviate the error by using conservative time interval bounds of temporal events (§5.3). Second, if multiple PLC processes run simultaneously, we still generate one combined PLC execution model by merging their variable sets and code logic, following the technique in *SCADMAN* [4]. As proved in *SCADMAN*, the merging will preserve the functionality of the original processes if the PLC cycle time is negligible and there is no race of variable updates. The race in PLC variable writes should be avoided to ensure robustness.

**Generation**. The PLC execution model is automatically extracted by static program analysis on PLC code. First, we need to translate the PLC code into programs that can be executed on general operating systems without PLC hardware, for the convenience of further analysis. Using Turing-complete languages to represent the control logic of PLC code has been well studied by previous work [25, 40, 56], where the event-driven PLC diagrams are mapped to if-then blocks of high-level programming languages. For instance, *TSV* [40] translates PLC code (*i.e.,* instruction lists) to a domain-specific language considering PLC-specific features such as timers. *SCADMAN* [4] leverages open-source compiler *MATIEC* to compile PLC code to C and further propose a method to merge the code from multiple PLCs. We borrow the above existing techniques to implement the code translation considering multiple PLCs. Then transforming the generated program into PLC execution model is straightforward. Each variable in the program is a system state

variable and the function in the PLC execution model is defined by the program itself. Details are in Appendix A.

**About configuration parameters**. From the survey of ICS testbeds [27], there are three types of configuration parameters in PLC code. (1) Parameters pushed to remote devices/machines, *e.g.,* CNC motor frequency in the motivating example. These parameters affect the physical processes outside PLCs and their impact will be captured by temporal events. (2) Timeout values of timers in PLC code. These parameters directly affect temporal relations thus they are modeled in temporal events as well. (3) Parameters as conditions. Developers use configuration parameters to enable or disable a specific function in the system. In this case, we regard each combination of these parameters as a unique version of the ICS and apply our system modeling method separately to each version.

## 5.3 Temporal events

Temporal dependencies are common in the execution of ICSs caused by both the control logic and physical dynamics of the environment. To model the temporal dependencies, we define events. An event is a triple of one *trigger*, one *operation*, and one *interval bound*. The trigger is one constraint on the system state. Given a system state, the event is triggered if the constraint is satisfied. The operation is a function updating one system state to another one. The operation has a causal relationship with the trigger. The interval bound indicates how much the time between the trigger and the operation is. Note that the bound can be real numbers or functions of symbolic configuration parameters. In the motivating example, for the event of CNC processing, the trigger is that the part is placed in the CNC, the operation is setting `CNC_Done` to positive, and the interval bound depends on the configuration of CNC motor frequency.

Moreover, we consider a special type of event called **interruptible events** whose operation only takes place when the constraint specified by the trigger is always satisfied during the whole time interval. Otherwise, the event is interrupted and the operation will not happen. One typical interruptible event is the timers implemented in PLC code. Instruction `TON` (Timer ON) enables the counting behavior of a timer. If `TON` is executed in consecutive cycles, the timer normally increases its value until reaches a threshold. Once `TON` is not executed in any cycle, the timer is reset. For this case, we define an interruptible event whose trigger is the activation of `TON`, the interval is the timer threshold, and the operation is setting PLC code variable `timer.DN` (Timer Done) to positive.

**Generation**. We semi-automatically generate temporal events with the assistance of developers. A set of interruptible events are first directly extracted from timers in PLC code. We extract other events from real-world PLC data traces by adopting the data mining technique used in VetPLC [56]. We collect 10-hour data traces of the PLCs for each testbed (recording values of PLC variables on each scan cycle), including the execution of the ICS under various configuration settings, and the data mining includes two main steps. First, it identifies the trigger and operation of events according to the causal relation. For each device connected to the PLC, updates on its input signals can be event triggers while updates on its output signals can be event operations. PLC code labels the connection between PLC variables and devices so this process can be automatic. Second, we parse all available data traces, identify the timestamp of all triggers and operations, and get the maximum/minimum time intervals for each statistically correlated trigger-operation pair. More details of data mining is in Appendix B.

**About configuration parameters**. If the temporal bound is affected by any configuration parameter, the temporal bound should be a function of configuration parameters instead of constant real numbers. We first split the data traces *w.r.t.* different configuration settings and mine the constant temporal bounds separately. We then apply regression analysis which regards configuration parameters as features and the interval bounds as outcome variables. Thereby, the regression fits a function of configuration parameters as temporal bounds.

The data mining is the best effort to alleviate the manual effort. However, it is not verified and requires the ICS developer to manually make complements if necessary. First, data mining only works on variables defined in PLC code (*i.e.,* sensor signals and in-memory variables in §5.1). Some causal relations involving physical processes, *e.g.,* conveyor delivers part, are not indicated by PLC code thus they cannot be automatically identified by data mining. However, they can be easily identified by developers who have knowledge of the workflow of the system. Second, when the volume of data traces is not sufficient to explore extreme executions, we need to consider a conservative interval bound. If necessary, developers need to broaden the bounds of event intervals according to the specification of physical devices and their expert knowledge. After the confirmation from developers, the bound should be further broadened by one PLC scan cycle time to alleviate the inaccuracy mentioned in §5.2. For instance, we broaden the interval bound from $[C_1, C_2]$ to $[C_1 - \tau, C_2 + \tau]$, where $0 \leq C_1 \leq C_2$ are constant real numbers and $\tau$ is the maximum PLC cycle time. In this way, even though two PLCs do not execute their scan cycle at the same time, all possible event orders are captured.

Note that the PLC code (or corresponding data traces) used in §5.2 and §5.3 is exactly the same. Therefore, the generated PLC execution model and temporal events are naturally compatible – they are functions on the same set of system state variables.

## 5.4 Symbolic Event Sequence Graph

Symbolic Event Sequence Graph (SESG) depicts possible execution traces of one ICS. The graph contains symbolic time intervals and symbolic configuration parameters so that we can use SMT tools to formally reason the configuration's impact on ICS execution. The symbolization is the main difference between SESG and existing ICS formal models. TEG model from TSV [40] does not symbolize time intervals thus cannot handle specifications with quantified time variables; TEG, TECG (from VetPLC [56]), and timed automata (used by UBIS [26] *etc.*) all regard configuration parameters as concrete fixed values so that they cannot model dynamic configurations. With the symbolic representation, the finite graph of SESG can concisely represent infinite concrete execution traces under all possible configuration settings.

**Formal definition.** First, we show part of the SESG related to the motivating example (§3.1) in Figure 3. The SESG is a directed graph $G = (V, E, \alpha, \beta, \gamma)$ over the set of system states $S$, a set of time intervals $T$, and a set of constraints $C$ on $T$. $V$ is the vertices of SESG and $\alpha : V \rightarrow S$ is a labeling function that maps vertices to system states. $E \subseteq V \times V$ is the edges of SESG. The labeling function $\beta : E \rightarrow T$ associates each edge to one non-negative symbolic time

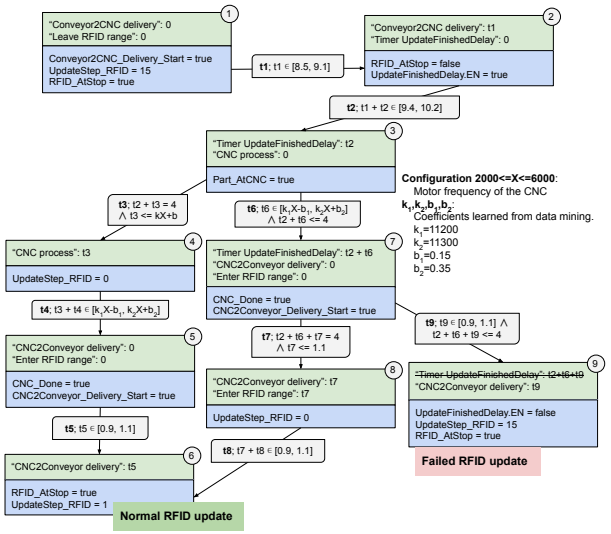



**Figure 3: SESG of the motivating example (§3.1). Vertices contain system states (selected variables, in blue) and triggered events (in green). Edges contain symbolic time intervals and guard constraints.**

interval while the labeling function $\gamma : E \rightarrow C$ associates each edge to one guard, which is a constraint on symbolic variables (*i.e.,* time intervals and configuration parameters). A *path* $\pi = (v_0, v_1, \dots)$ where $v_0, v_1, \dots \in V$ is a sequence of system states connected by edges. The path represents one execution of the ICS, which starts from one initial state $v_0$ and shifts system states along the edges on the path. Moreover, the execution is feasible only when the *path constraint*, which is the intersection of all guards along the path $(\bigwedge_i^{v_i, v_{i+1} \in \pi} \gamma(v_i, v_{i+1}))$, is satisfied.

The example graph in Figure 3 starts from the system state where the part delivery from the conveyor to the CNC is triggered (vertex #1) and ends when the part enters the RFID detection range (vertices #6, #9). There are three different paths: (123456), (123786), and (12379) (the numbers are vertex IDs). Particularly, path (12379) demonstrates a failed RFID update since the `UpdateStep_RFID` is not reset at #9. During verification and parameter recommendation, we can use the path constraint, which is associated with symbolic time intervals $t_1, t_2, t_6, t_9$ and symbolic configuration $X$ (motor frequency), to analyze which $X$ can cause the violation. In contrast, the other two paths are safe executions since they reset the RFID status at #4 and #8 respectively before reaching #6.

**Graph construction**. Algorithm 1 is the algorithm of SESG construction. Initially, there are a set of initial vertices indicating the initial system states. Then the algorithm executes in loops and in each iteration we try to connect new vertices to the graph. To implement the algorithm, we introduce several data structures. First, each vertex records its triggered events (green boxes in Figure 3) and accumulated time since they are triggered (*i.e.,* triggered time). For instance, the vertex #1 in Figure 3 is associated with the system state where delivery from the conveyor to the CNC is newly started. The green box records the triggered events including "Conveyor2CNC delivery" and "Leave RFID range" and the triggered time for both events is zero since they are newly triggered by the current system state. We use the map structure $M$ to represent

**Algorithm 1:** Algorithm of constructing SESG.

**Input:** Vertices $V_0$ (initial states), PLC execution Exec, event set $\Omega$, system state set $S$, constraint set $C$.
**Output:** SESG $G = (V, E, \alpha, \beta, \gamma)$.

1 Initialize SESG $G$: $V \leftarrow V_0$, taskStack $\leftarrow V$;
2 Initialize event map $M : V \times \Omega \rightarrow \mathbb{R}_{\geq 0} \cup \{-1\}$;
3 Initialize invariant map $I : V \rightarrow C$, $I(v_0) = true$;
4 **while** $v_c \leftarrow$ *taskStack.pop* **do**
5  **for** $\omega \in \Omega$ **do**
6  Get Trigger from $\omega$;
7  **if** *Trigger*$(v_c)$ *and* $M(v_c, \omega) = -1$ **then**
8  $M(v_c, \omega) \leftarrow 0$;
9  **end**
10  **if** *not* *Trigger*$(v_c)$ *and* $\omega$ *is interruptible and* $M(v_c, \omega) \geq 0$ **then**
11  $M(v_c, \omega) \leftarrow -1$;
12  **end**
13  **end**
14  **for** $\omega \in \Omega$ *and* $M(v_c, \omega) \geq 0$ **do**
15  Get $(t_{min}, t_{max})$ and Operation from $\omega$;
16  $v'_c \leftarrow$ Operation$(v_c)$;
17  **while** $v'_c \neq$ *Exec*$(v'_c)$ **do**
18  $v'_c \leftarrow$ Exec$(v'_c)$;
19  **end**
20  $t \leftarrow$ Symbolic$(\mathbb{R}_{\geq 0})$;
21  $g \leftarrow M(v_c, \omega) + t \in [t_{min}, t_{max}]$;
22  **for** $\omega' \in \Omega - \{\omega\}$ *and* $M(v_c, \omega) \geq 0$ **do**
23  $M(v'_c, \omega') \leftarrow M(v_c, \omega') + t$;
24  Get interval $(t'_{min}, t'_{max})$ from $\omega'$;
25  $g \leftarrow g \wedge M(v'_c, \omega') \leq t'_{max}$
26  **end**
27  $inv \leftarrow I(v_c) \wedge g$;
28  **if** *SmtCheck*$(inv)$ *= sat* **then**
29  **if** $\exists v_d \in V, v'_c = v_d$ **then**
30  $v'_c \leftarrow v_d$;
31  **else**
32  Add $v'_c$ to $V$, add $v'_c$ to taskStack;
33  **end**
34  $I(v'_c) \leftarrow I(v'_c) \vee inv$;
35  Add $(v_c, v'_c)$ to $E$, $\beta(v_c, v'_c) \leftarrow t$, $\gamma(v_c, v'_c) \leftarrow g$;
36  **end**
37  **end**
38 **end**

this mapping from vertices to triggered events and triggered time. Second, each vertex has an invariant. The system state is reachable when the corresponding invariant is satisfiable. The invariants of initial vertices are set to constant true. In each loop iteration, the algorithm picks one vertex ($v_c$) that may have child vertices and does the following steps.

First, check whether the vertex triggers new events (line 5-12). For each defined temporal event, if the trigger constraint is satisfied by the current system state and the event is not yet triggered, add the new event to $M$ and set the triggered time to zero. Especially, interruptible events that are already triggered on the current vertex should be removed if their trigger constraints are not satisfied. After this update, we have the event list that is triggered but not yet finished on the current system state $v_c$. For instance, if the $v_c$ is the vertex #1 in Figure 3, the event "Conveyor2CNC delivery" is triggered because the trigger constraint `Conveyor2CNC_Delivery_Start=true` is satisfied.

Second, generate new vertices that can be the next system state after $v_c$ (line 15-26). The emergence of new system states is accompanied by the operation of events. Once an event is finished, its operation functions will directly modify the system state and the PLC execution will further update more system state variables. Corresponding to each triggered event on $v_c$, notated by $\omega$, the construction of the next vertex $v'_c$ is done by the following two steps. (1) The algorithm first executes the operation function defined by the

event $\omega$ and then repeatedly executes the PLC execution until the system state is stable. The final stable system state is assigned to $v_c'$. Note that we abstract PLC execution cycle intervals as negligible, as discussed in §5.2. (2) The algorithm updates the triggered events on $v_c'$ ($M(v_c')$), which is copied from $M(v_c)$ but has following updates. $\omega$ is removed because it is finished at $v_c'$. The triggered time of other triggered events increases by the time interval of edge between the previous vertex and the new vertex ($\beta(v_c, v_c')$).

Third, check the validity of the state transition considering temporal constraints (line 27-36). We first calculate the guard of edge from $v_c$ to $v_c'$ by combining the two following constraints. (1) The finished event $\omega$ must be finished within the range defined by the event's interval bounds. This constraint can be written as $M(v_c, \omega) + \beta(v_c, v_c') \in [t_{min}, t_{max}]$ where $[t_{min}, t_{max}]$ is the interval bounds of event $\omega$. (2) Other triggered but unfinished events are possible to finish after $v_c'$. In other words, the triggered time of triggered events on $v_c'$ cannot exceed the their maximum interval bounds. Formally, the constraint is $\bigwedge_{\omega'}^{M(v_c', \omega') \geq 0} M(v_c', \omega') \leq t_{max}^{\omega'}$, where $t_{max}^{\omega'}$ is the maximum interval of event $\omega'$. Then we calculate the invariant of the new vertex $v_c'$, which is the intersection of the previous vertex $v_c$'s invariant and the guard of edge $(v_c, v_c')$. If the invariant of $v_c'$ is satisfiable, there exists a solution of symbolic variables (*i.e.,* edge intervals and configurations) making that at least one path from initial vertices to $v_c'$ satisfies temporal constraints. We add $v_c'$ into the graph in this case. In addition, we disallow duplicated vertices for minimizing the graph size. We consider two vertices equivalent when their system states and triggered events are the same. If the new vertex $v_c'$ is already in the graph, we take a union of their invariants without adding any vertex.

**Validation.** We use the real-world data traces to validate SESG. SmtConf can automatically examine whether each data trace is correctly represented, *i.e.,* at least one path in SESG reproduces the execution. If not, developers are involved to debug the inaccuracy (*e.g.,* adding or fixing events, as mentioned in §5.3) until the validation passes. It is an extremely hard challenge to guarantee the perfection of the model due to the lack of ground truth but our system modeling achieves the best-effort accuracy thanks to the feedback loop of validation. In other words, SESG ensures 100% coverage of possible execution identified by developers or data traces, which is the best knowledge a defense system can have.

# 6 RUNTIME MITIGATION

Based on SESG, SmtConf detects safety hazards and recommends safe parameters. We introduce specifications (§6.1), offline and online processes (§6.2, §6.3), and one end-to-end example (§6.4).

## 6.1 Specification

SmtConf's specification adopts the grammar of Timed Propositional Temporal Logic (TPTL), as defined in Definition 1. TPTL not only reasons the order of events but also measures the time between two events. TPTL and its variants are commonly used in model checking and are effective in checking temporal properties [6].

**Definition 1** (TPTL formula)**.** Let $P$ be a propositional system containing a set of atomic logical proposition symbols $\{p_1, p_2, \ldots, p_{|A|}\}$, and let $\Sigma = 2^A$ be a finite alphabet composed of propositions in $P$. TPTL formula is defined by the grammar: $TPTL \ni \phi ::= p \mid \phi \wedge \phi \mid \neg\phi \mid \phi \boldsymbol{U} \phi \mid x \sim c \mid x.\phi$ where $p \in P$, $x$ ranges over a finite set of clock variables, $c$ ranges over $\mathbb{Q}$, and $\sim \in \{\leq, <, =, >, \leq\}$.

Safety properties and liveness properties matter for industrial systems and we can use TPTL to define them. For instance, safety property $\square(\neg Danger)$ means that the safety hazard defined by formula $Danger$ never happens, and liveness property $\diamond x.(Finish \wedge x < 60)$ means that the system reaches a state of $Finish$ eventually within 60 time units (seconds). In Table 2, we list safety specifications used in our experiments.

Specification checking is applied on SESG paths. According to the semantic of specifications, it generates a violation constraint (an SMT expression) for each path and any solution to the constraint is a concrete execution path violating the specification. For instance, we can check the path (1,2,3,7,9) of the motivating example (Figure 3) against the specification about "skipped RFID update" (#2 in Table 2). The violation constraint is a constant *true* because the vertex #9 will surely cause a violation. To make the specification checking deterministic, the length of checked paths must be finite. Infinite paths are possible in SESG because of the existence of loops. However, since ICS tasks have hard time limits, the infinite paths surely violate the basic liveness requirement. For example, the SMART cell requires the part processing to finish within 60 seconds (specification #1 in Table 2) so that paths longer than 60 seconds or paths with back loops are violation paths.

## 6.2 Offline Safety Constraint Generation

We apply specification checking (§6.1) on each SESG path and calculate a safety constraint making all violations impossible. Formally, we notate the SESG as $G = (V, E, \alpha, \beta, \gamma)$, the set of possible paths as $\Pi$, the set of specifications as $S$, and the set of symbolic configurations as $W$. Especially, we use $T^\pi$ to represent the set of symbolic time intervals along the path $\pi \in \Pi$. As mentioned in §5.4, one path $\pi \in \Pi$ is possible to happen when the path constraint $C_p^\pi = \bigwedge_i^{v_i, v_{i+1} \in \pi} \gamma(v_i, v_{i+1})$ is satisfiable. As mentioned in §6.1, one path $\pi \in \Pi$ violates at least one specification when the violation constraint $C_s^\pi = \bigvee_{s \in S} s(\pi)$, where $S$ is the set of safety specifications, is satisfied. Given the facts that one path is safe iff no specification is violated whenever the path is possible and the system is safe when all paths are safe, the safety constraint for the ICS is a constraint on symbolic configurations $W$, which can indicate not only safety range of a single parameter but also complex interaction of multiple parameters. The constraint is:

$$C_{safe} = \bigwedge_{\pi \in \Pi} \forall T^\pi \ (C_p^\pi \implies \neg C_s^\pi). \tag{1}$$

Generating the above constraint using SMT tools such as Z3 [17] may be time-consuming since the formula contains quantifiers and iterates all paths in SESG. We introduce a few implementation tricks to alleviate the time cost in Appendix C.

## 6.3 Online Detection and Recommendation

**Detection**. Given configuration parameters and their concrete values, SmtConf checks whether the configuration can violate any safety specifications. SmtConf builds a configuration constraint (denoted by $C_W$) that restricts each symbolic configuration parameter to be equal to the concrete value. The ICS is verified to be violation-free when the safety constraint can never be violated under the specific configuration (*i.e.,* Equation 2 is not satisfiable).

**Table 2: Potential violations and corresponding specifications for ICS systems.**

| ID | Name | ICS | Description | Safety specification to avoid the violation (TPTL) |
|---|---|---|---|---|
| 1 | Unfinished process | SMART cell | The part processing does not finish in 60 sec. | $\Diamond x.(x < 60 \wedge Pallet\_Status = PASSED\_STOP \wedge Part\_Status = ON\_PALLET)$ |
| 2 | Skipped RFID update | SMART cell | RFID update is failed. | $\Box((\neg RFID\_AtStop \wedge \Diamond x.RFID\_AtStop) \implies \Diamond y.(y < x \wedge UpdateStep\_RFID = 0))$ |
| 3 | Blocked stop | SMART cell | The stop fails to release the pallet in 30 sec. | $\Box x.(Pallet\_Status = AT\_STOP \implies \Diamond y.(y - x < 30 \wedge Pallet\_Status = PASSED\_STOP))$ |
| 4 | Robot in danger zone | SMART cell | The robot fails to return its safe zone. | $\Box x.(Delivery\_Start \implies \Diamond y.(y - x < 30 \wedge Robot\_Go\_Home))$ |
| 5 | Repeated delivery | SMART cell | The robot triggers duplicated delivery tasks. | $\Box(\neg(Part\_Status = PROCESSED \wedge Conveyor2CNC\_Delivery\_Start))$ |
| 6 | Pallet-gate collision | SMART cell | The holdback gate hits the pallet. | $\Box(\neg(Pallet\_Status = PASSING\_HOLDBACK \wedge Holdback\_Extend))$ |
| 7 | Wrong diversion | SMART diverter | Parts go to wrong route. | $\Diamond x.(x < 40 \wedge Pallet1\_Status = EXIT1 \wedge Pallet2\_Status = EXIT2)$ |
| 8 | Pallet collision | SMART diverter | Two consecutive pallets collide. | $\Box(Pallet1\_Status \neq Pallet2\_Status)$ |
| 9 | Unfinished process | Fischertechnik | The process does not finish in 40 seconds. | $\Diamond x.(x < 40 \wedge Part1\_Status = EXIT \wedge Part2\_Status = EXIT)$ |
| 10 | Ram-part collision | Fischertechnik | A ram pushes without any part in the ram. | $\Box(Part\_Status = ENTERING\_RAM \implies \neg Ram\_Push))$ |
| 11 | Part collision | Fischertechnik | Two consecutive parts collide. | $\Box(Part1\_Status \neq Part2\_Status)$ |
| 12 | Unfinished CNC | Fischertechnik | The CNC releases unfinished parts. | $\Box x.(Part\_Status = ENTER\_CNC \implies \Diamond y.(y - x > 3 \wedge Part\_Status = LEAVING\_CNC))$ |
| 13 | Fast ram | Fischertechnik | The ram pushes the part too hard. | $\Box x.((Part\_Status = AT\_RAM \wedge Ram\_Push) \implies \Diamond y.(y - x > 0.5 \wedge Part\_Status = PASSED\_RAM))$ |

$$C_W \wedge \neg C_{safe} \tag{2}$$

**Recommendation**. Given a search range of configuration parameters (denoted by $C_W$) and an optimization objective (a function of configuration parameters, denoted by $f(W)$), SmtConf generates concrete values for configurations that are violation-free, within the range, and optimized towards the objective function. The recommendation algorithm first prepares a configuration constraint indicating the search range of safe configuration values. The recommendation is done by solving the optimization problem as defined in Equation 3.

$$\begin{aligned} \min \quad & f(W) \\ \text{s.t.} \quad & C_W \wedge C_{safe} \end{aligned} \tag{3}$$

**Correctness**. As mentioned in §5.4, SESG exhaustively include all possible execution paths assuming sound model inputs (e.g., events). Based on the complete SESG, the safety constraint by definition excludes any safety violation defined by the specifications. As a result, under the same assumption as the correctness of SESG, the detection achieves zero false negatives, and the recommendation guarantees that the recommended parameters violate no specifications. However, false positives of detection are possible theoretically since temporal events have conservative time bounds and consider unexpected failures that may not happen in the real world.

## 6.4 End-to-end Mitigation Process

In this section, we demonstrate the execution of SmtConf on the motivating example (§3.1).

Before the ICS starts running, the model SESG is built and the safety constraint is generated. During the generation of safety constraint, the configuration parameters are symbolic variables restricted by their maximum ranges, *e.g.,* CNC motor frequency (notation as $X$): $2,000 \leq X \leq 6,000$.

During runtime of the ICS, when SmtConf observes the change of the CNC's motor frequency from 3,000 rpm to 5,000 rpm, violation detection is triggered where the configuration constraint is $X = 5000 \wedge \ldots$ (other parameters are omitted). The detection process finds the configuration constraint violates the safety constraint and rejects the new configuration.

The operator intends to increase the motor frequency to increase production but his attempt has failed. He can command SmtConf to launch the recommendation process to determine an optimized parameter choice. SmtConf by default sets the configuration constraint to $3000 \leq X \leq 5000 \wedge \ldots$ where $X$ is between the parameter values before and after the change and other parameters remain equal to their current values. Also, SmtConf by default sets the objective function to $|X - 5000|$ which optimizes $X$ to the original target of configuration change. In the above settings, the optimization process generates the optimized configuration $X$=3,814 rpm.

**Table 3: ICSs and corresponding SESG.**

| | System complexity | | | Model (SESG) generation | | | | Safety constraint(s) |
|---|---|---|---|---|---|---|---|---|
| ICS | #Cfg | #Var | #Event | #node | #edge | #path | time(s) | |
| SMART cell | 7 | 42 | 26 | 4748 | 6370 | 3504 | 164.85 | 1331.94 |
| SMART diverter | 5 | 24 | 14 | 233 | 307 | 130 | 66.13 | 168.17 |
| Fischertechnik | 5 | 35 | 27 | 6026 | 7688 | 2509 | 1280.74 | 288.70 |

# 7 EVALUATION

We evaluate SmtConf in three aspects: (1) *correctness* of detection and recommendation, (2) *efficiency* – time cost of SmtConf processes, and (3) *automation* – amount of required manual efforts. To answer the above questions, we construct 18 test cases from SMART and Fischertechnik (§7.2) and evaluate SmtConf's components respectively. We conduct all the experiments on a server with Intel Xeon CPU E5-4620v2 2.60GHz. Note that the experiments of detection and recommendation are on SESG models which are validated using data traces and expert knowledge. We discuss the manual effort in model validation separately in §7.6.

## 7.1 Implementation

SmtConf is implemented in 7K LOC in total including 3,220 lines of Closure/Java code for program analysis, 1,859 lines of Python code for data mining, and 1,152 lines of Python code for the core algorithms (SESG, detection, recommendation, *etc.*). We use SMT solver Z3 for solving constraint satisfaction and optimization problems.

Besides SmtConf, we use simulators of tested ICS testbeds, which are implemented and validated by ICS developers, for evaluation purposes. Given a system configuration, the simulator updates the status of all ICS objects (*e.g.,* sensors, PLCs, robots, parts/pallets) with a fixed interval of 10 ms according to the control logic and physical laws, following the state-of-the-art best practices [16]. Our simulator also introduces randomization to model the uncertainty of sensors and actuators.

## 7.2 Experimental Setup

**Systems**. We test SmtConf on three ICSs listed in Table 3. (1) Part processing in one SMART cell, as discussed in §3.1. (2) Parts passing one diverter with two exits in SMART. The task of the diverter is to route the parts to their correct destination according to their RFID status. (3) Part processing in Fischertechnik. Fischertechnik is a testbed of a manufacturing system and its part processing is finished when the part reaches the exit and has been processed by two CNCs. More details about SMART diverter and Fischertechnik are presented in Appendix D. Table 3 shows the complexity of the scenario (*i.e.,* count of configurations, state variables, and events).

**Table 4: Test cases and evaluation results of SmtConf's verification and recommendation.**

| ID | ICS | Improper configuration parameter | Violated specification | Verification | | Recommendation | |
|---|---|---|---|---|---|---|---|
| | | | | Correct | Time(s) | Value (error) | Time(s) |
| 1 | SMART cell | N/A | N/A | √ | 0.40 | N/A | N/A |
| 2 | SMART cell | CNC2Conveyor_Deliver_Finish signal timeout: 10 s → 5 s | Blocked stop, unfinished process | √ | 0.41 | 5.81 s (+1.72%) | 0.43 |
| 3 | SMART cell | Part_AtCNC signal timeout: 3 s → 1.4 s | Robot in danger zone | √ | 0.40 | 1.13 s (+2.67%) | 0.42 |
| 4 | SMART cell | CNC's motor frequency: 3,000 rpm → 5,000 rpm | Skipped RFID update | √ | 0.41 | 3,814 rpm (-3.39%) | 0.43 |
| 5 | SMART cell | ReleasingMachinedPart timer threshold: 8s → 5s | Repeated delivery, unfinished process | √ | 0.41 | 5.91 s (+1.60%) | 0.43 |
| 6 | SMART cell | Conveyor speed: 500 mm/s → 225 mm/s | Pallet-gate collision | √ | 0.40 | 385 mm/s (+3.08%) | 0.43 |
| 7 | SMART cell | ReleaseHoldback timer threshold: 0.4 s → 0.15 s | Pallet-gate collision | √ | 0.40 | 0.201 s (+0.50%) | 0.42 |
| 8 | SMART cell | Conveyor speed: 500 mm/s → 165 mm/s | Pallet-gate collision, unfinished process | √ | 0.40 | 385 mm/s (+3.08%) | 0.42 |
| 9 | SMART diverter | N/A | N/A | √ | 0.04 | N/A | N/A |
| 10 | SMART diverter | Pallet interval: 8 s → 6 s | Wrong diversion | √ | 0.04 | 6.76 s (+2.54%) | 0.12 |
| 11 | SMART diverter | WaitAtHoldback timer threshold: 3 s → 5 s | Pallet collision | √ | 0.04 | 4.5 s (+0.00%) | 0.12 |
| 12 | SMART diverter | Conveyor speed: 500 mm/s → 385 mm/s | Wrong diversion | √ | 0.05 | 520 mm/s (+2.84%) | 0.09 |
| 13 | Fischertechnik | N/A | N/A | √ | 0.35 | N/A | N/A |
| 14 | Fischertechnik | TransitingToRam timer threshold: 2 s → 0.9 s | Ram-part collision | √ | 0.35 | 0.934 s (+3.83%) | 0.34 |
| 15 | Fischertechnik | Machining timer threshold: 3 s → 2 s | Unfinished CNC | √ | 0.35 | 2.432 s (+1.34%) | 0.45 |
| 16 | Fischertechnik | Part interval: 8 s → 6 s | Part collision | √ | 0.36 | 7.520 s (+0.67%) | 0.35 |
| 17 | Fischertechnik | Ram speed: 100% → 166% | Fast ram | √ | 0.36 | 125% (+0.00%) | 0.36 |
| 18 | Fischertechnik | Conveyor speed: 40 mm/s → 30 mm/s | Unfinished process | √ | 0.36 | 36 mm/s (+4.32%) | 0.38 |

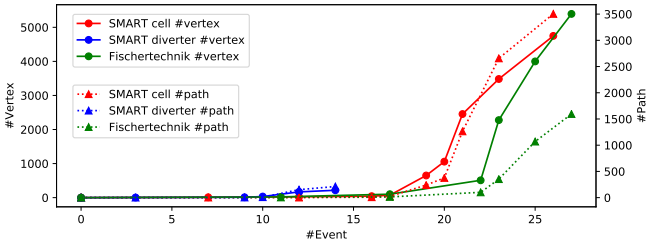



**Figure 4: The scalability of system modeling: number of vertices and paths *w.r.t.* count of temporal events.**

**Specifications**. We create TPTL-based safety specifications for each scenario as listed in Table 2. The specifications require the ICS to finish its task in time and avoid unexpected accidents such as collisions, blocked processes, etc.

**Test cases**. We select in total 18 test cases that may happen on the 3 ICSs, as listed in Table 4. 3 out of 18 test cases have safe configurations and SmtConf is expected to report the non-existence of violations. In the other 15 test cases, the adversary improperly tunes one parameter. The task of SmtConf is to detect the wrong parameter and recommend a safe value of the same parameter.

## 7.3 System Modeling

**Efficiency**. We record the complexity of the generated SESG as well as the execution time of system modeling in Table 3. Indicated by the algorithms, the execution time of graph construction and safety constraint generation increases linearly as the number of vertices and paths in SESG increases, respectively. Among the three ICSs, Fischertechnik has the largest graph (6K vertices and 7K edges) and the longest execution time of graph construction (about 20 minutes) while SMART cell has the most paths (3.5K) and the longest execution time of safety constraint generation (about 22 minutes). The whole offline process takes 4-25 minutes.

**Scalability.** Since the performance of SmtConf is closely associated with the SESG size, we analyze how the graph size increases when the ICS gets more complex. Intuitively, one key factor is the number of events and the interaction among them. If the order of two events is not deterministic, the path will be forked to two branches so that the size of the graph increases. Generally, the chance of this forking is higher when there are more intertwined events. To confirm this hypothesis, we regenerate the graphs for three ICSs with selected subsets of events, and each event subset models part of the original ICS task. We present the relation between graph complexity and the number of events in Figure 4. Generally, more events result in a larger graph and the modeling of the complicated part of ICS tasks grows the graph significantly. For SMART cell, the graph complexity suddenly grows when the number of events increases from 20 to 22. The newly added 2 events are about a complicated process that the part is being placed to the conveyor along with a simultaneous RFID update. Also, we found that the graph grows faster when the graph is larger, which is between linear growth and exponential growth. For better efficiency, we recommend the developers to split ICS tasks into independent stages and apply SmtConf to each stage respectively.

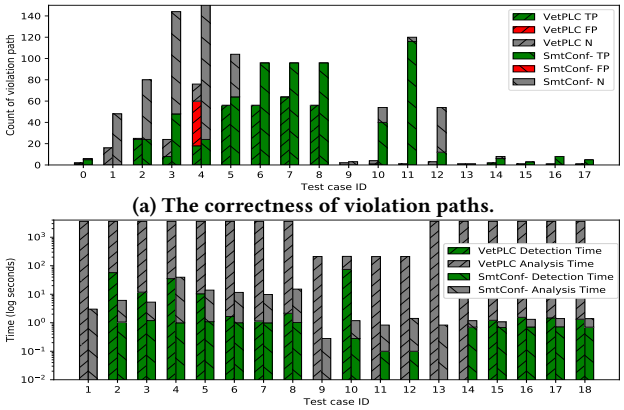
## 7.4 Detection

**Correctness**. We apply SmtConf's detection on the 18 test cases in Table 4. 3 test cases have safe configurations and SmtConf successfully validates that their configurations are compliant with specifications. The other 15 of them contain injected improper configurations and SmtConf reveals the violations. We also use simulation to validate the detection results. We run each test case in the simulator repeatedly 1,000 times and the simulation can reproduce all violation scenarios. So SmtConf shows 100% true positive rate and 0% false positive rate on the 18 cases.

**Efficiency**. Since the safety constraint is generated offline, the online detection is a lightweight constraint satisfaction solving process that takes at most 0.45 seconds among 18 cases. Because of the same reason, the time cost mainly depends on the complexity of the safety constraint and thus is almost the same for test cases under the same ICS. The real-time violation detection leaves attackers no room to use improper configurations to damage the ICSs.

**Comparison with VetPLC** [56]. VetPLC is a state-of-the-art tool for discovering safety problems in PLC systems but can only check one specific configuration setting in each execution. To make a fair comparison of verification capability, we produce a downgraded version of SmtConf called SmtConf- where configuration parameters are no more symbolic variables but concrete numbers. On each test case, SmtConf- generates SESG and directly checks each path against specifications to reveal violations, which is a similar workflow to VetPLC's. Both tools are applied to the 18 cases in Table 4 and we aim to evaluate the different efficiency and accuracy of SESG and VetPLC's model TECG.

*Comparison of correctness.* We manually check whether paths found by either SmtConf- or VetPLC represent feasible execution in the real world (we translate VetPLC's "event sequences" to the




(a) The correctness of violation paths.

(b) The efficiency of violation detection.

**Figure 5: The comparison of violation detection between SmtConf- (downgraded SmtConf) and VetPLC.**

format of SESG paths). A violation path is a true positive if the execution path obeys the PLC code logic and physical laws. The result is shown in Figure 5a.

First, SmtConf- covered all execution paths discovered by VetPLC and analyzed 216% more paths than VetPLC. No false negatives are observed in SmtConf-'s results while VetPLC missed 354 (55% of total) true violation paths and failed to detect violations in case #11 and #12. There are two reasons for the difference. First, VetPLC reasons the order of events by discretizing the time domain and sampling a subset of possible time intervals. In contrast, SESG represents time using symbolic real numbers in the continuous-time domain, which guarantees 100% coverage on possible execution paths (assuming the event definition is sound). Second, due to the discretization approach, VetPLC has scalability problems that 13 out of 18 test cases are not finished before time-out (one hour), which results in more missing paths.

Second, VetPLC has 42 false positives (*i.e.,* infeasible violation paths) which do not exist in SmtConf-. This is because of the imprecise system model adopted by VetPLC. Instead of handling PLC execution, uninterruptible events, and interruptible events (two categories of events defined in §5.3) separately as SESG does, VetPLC's model handles all behavior of ICSs as uninterruptible events. When applying VetPLC on the motivating example (case #4 in Table 4), the timer `UpdateFinishedDelay` continues counting after part enters the RFID range and the timer is disabled, which results in infeasible paths.

*Comparison of performance.* For both SmtConf- and VetPLC, the execution time of verification-based detection mainly depends on the size of SESG and the algorithm's processing speed. In terms of processing speed, SmtConf- generates 13 paths per second while VetPLC generates 11.5 paths per second. However, 99.4% of paths generated by VetPLC are duplicated paths while all paths from SmtConf- are unique. Two paths are duplicated means that they present the same order of events though the exact time intervals may differ, and Figure 5a has removed duplication.

As shown in Figure 5b, we evaluate two metrics of latency: (1) detection latency which is the time used to find the first violation path; (2) analysis latency which is the consumed time for enumerating all possible paths. In terms of the full analysis latency, SmtConf- finishes each test case in at most 40 seconds while VetPLC failed

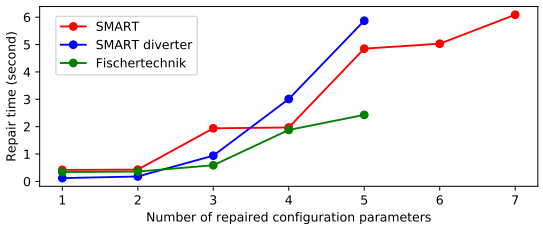



**Figure 6: Execution time of recommendation of parameters.**

to finish the same test cases in one hour (we set timeout as one hour). For the detection latency, SmtConf- takes no more than 1.3 seconds while VetPLC takes more than 10 seconds on complicated cases. SmtConf- significantly reduces the execution time because SmtConf- uses SMT theorem proving to solve possible paths from formulas instead of blind sampling.

In summary, besides the support of analyzing safe configuration ranges, SmtConf's fully symbolic system modeling on the continuous-time domain achieves high coverage and high accuracy on violation detection.

## 7.5 Recommendation

For the 15 test cases with violations, SmtConf recommends safe values for the detected improper parameter, and the objective is set to minimize the distance to the original target value, as described in §6.4. We record the execution time and the optimized configuration values in Table 4.

**Correctness**. For each test case with violations, We manually calculate the best configuration choice as ground truth according to expert knowledge and confirm that the recommended values are within the safe range. In Table 4, we calculate the percentage error as (recommended value - ground truth). For instance, in the motivating example (test case #4 in Table 4), the motor frequency of about 3,950 rpm is the best safe choice for maximizing production. SmtConf recommends 3,814 rpm so that error is -3.39%. Table 4 shows that the error of recommendation is smaller than 5% on all test cases. One can alleviate the error by tuning the optimization algorithm, which is a trade-off between accuracy and performance.

Besides, we use the ICS simulator to validate SmtConf's recommendation results. We choose different configuration values within the range and use each choice to continuously simulate the ICS task 1,000 times. The best configuration found by the simulation is consistent with our manual analysis. The system patched by the recommendation process works reliably: 1,000 parts are correctly processed in SMART cell and Fischertechnik in on average 49 seconds and 35 seconds respectively while SMART diverter routes all 2,000 parts to correct destinations.

**Performance**. We use execution time to evaluate the performance of recommendation, as shown in Table 4. Generally, the recommendation finishes in only 0.5 seconds, thanks to the offline generated safety constraint. Similar to online detection, the recommendation time depends on the complexity of the safety constraint instead of the specific configuration parameter values. As a result, SMART cell requires a slightly longer recommendation time than the other two ICSs because of its high quantity of paths.

**Multi-parameter recommendation**. As mentioned in §6.3, the configuration constraint can be modified to recommend multiple parameters in one pass. To measure SmtConf's multi-parameter

recommendation performance, on each ICS (i.e., case #1, #9, #13 in Table 4), we repeat the recommendation on a various number of parameters. If we are making a recommendation for $k$ out of $N$ parameters, the configuration constraint restricts that the $k$ parameters are in their definition range and the others are equal to their original value. Correspondingly, we set the objective function as the maximum time cost for the ICS task to complete in order to maximize production throughput. As shown in Figure 6, though recommending multiple parameters increases the time cost, the recommendation is effective and scalable, which can optimize 7 parameters (all parameters in SMART cell) in 6 seconds.

### 7.6 Automation

The workflow of SmtConf is mostly automated except for part of event mining, specifications, and the optional customization of recommendation algorithm.

In system modeling, SmtConf's program analysis (§5.2) can accurately extract PLC execution from PLC code which requires no manual effort. The data mining (§5.3), however, is not guaranteed to be accurate and complete so we asked developers to manually examine the mined events. For the three tested ICS, data mining correctly extracts 49 out of 67 (80%) of total events. The developers added 13 events about physical dynamics (*e.g.,* robot arms move parts) as they are not defined by PLC variables, and modify the temporal bound of 5 events to cover some worst cases (*e.g.,* slow RFID updates). The developers spent on average two man-hours for each ICS to complement mined events. For detection and recommendation, specifications are manually written using TPTL grammar. The 13 specifications in Table 2 are written in about 30 minutes. For recommendation especially, developers can use the default configuration range and objective function as stated in §6.4 without any manual effort. Optionally, developers can modify the above two inputs with one line of code and a few seconds.

## 8 DISCUSSION

**Scope of attacks**. As discussed in the threat model (§3.2), we assume the internal communication is secure and PLC devices are protected. SmtConf aims at improper configuration injected at runtime instead of control logic injection or sensor spoofing. We emphasize that configuration manipulation is easier to launch but overlooked by existing defenses.

**Accuracy issues**. The algorithms for verification and recommendation assure their correctness under the assumption of a sound model which exhaustively covers the possible execution of the real-world ICS. However, the system modeling (§5) has three potential sources of inaccuracy and we alleviate the error separately. First, the data mining of temporal events may not be complete or accurate due to the nature of the statistical method and the limited size of collected data traces. We propose to use real data traces to validate the model to discover easy bugs and require the developers' manual effort to do the final examination. Second, the system model SESG ignores the interval PLC scan cycles. We alleviate the error by using preservative temporal bounds 5.3. Third, we borrow *SCADMAN*'s method [4] to combine code logic from multiple PLCs into one single function, which assumes zero race conditions in PLC processes. Developers can leverage existing code analysis methods [25] to detect such race conditions.

## 9 RELATED WORK

**ICS attacks.** A few studies explore strong and stealth attacks to challenge ICS security. To reduce the visibility of the attack from system operators, [21] proposes a man-in-the-middle attack where invaded PLC intentionally sends spoofed signals to monitoring tools. Other attacks on ICS including Denial-of-Sevice (DoS) [49] and control logic injection [55] also consider the similar signal spoofing method to hide attacks. To obtain the system model, some model inferring methods through data analysis [39] or deep learning on HMI screenshots [48] are proposed. Based on existing attacks, we define our threat model as §3.2.

**ICS defenses.** Most defenses focus on detecting attacks or potential safety problems. Some studies mine control invariants and check system behavior against the invariants to report possible attacks [4, 7, 16, 38, 54], and there exists a trade-off between false positives and false negatives. Verification-based defenses [51], on the other hand, though require more prior knowledge of system models, can provide stronger guarantees of detection accuracy. Some studies apply model checking tools on the PLC domain to verify PLC code [13, 14, 25, 40], but cannot reason the interaction between PLC and other system components. Further research work [41, 46, 56] add the physical/digital device into the system model. SmtConf takes one step further to consider the runtime safety hazards caused by dynamic parameters. Besides, [24] presents a device for inserting runtime formal verification processes into PLCs, which provides the hardware basis of SmtConf.

**Timed system repair.** Timed systems are formal models representing system behavior with temporal dependencies, which are often used to model real-time systems including ICS. Automatic repair mechanisms are proposed for various models including Petri net models [9, 10], timed automaton [32, 33], which eliminate violations of specifications by modifying the models. However, existing methods cannot be adapted to ICS parameter repair since they do not consider the dynamic configurations and customization of objectives. SmtConf's recommendation is the first practical algorithm to generate patches on ICS parameters.

## 10 CONCLUSION

We design and implement SmtConf, an automatic runtime mitigation framework for ICS misconfiguration vulnerabilities, to detect violations against safety specifications and automatically generate optimized safe parameters. SmtConf improves model-based verification techniques on ICSs by considering dynamic parameters that can be improperly modified at runtime. By analyzing the impact of dynamic configuration parameters, SmtConf is an efficient tool for guiding the developers to safely configure complicated ICSs.

## ACKNOWLEDGMENTS

This work is partially supported by NSF under the grant #2112562, CMMI-2038215, CNS-1930041, CNS-1544678, OAC-2115167 and DGE-2041960, DARPA HR00112120009 Cooperative Agreement, and a USHE Deep Technology Initiative Grant. The authors would like to thank anonymous reviewers and Dr. Yuru Shao for valuable feedback.

# REFERENCES


[1] 2019. Cyberattack Hits Indian Nuclear Plant. https://www.armscontrol.org/act/2019-12/news/cyberattack-hits-indian-nuclear-plant

[2] 2019. Toyota to Close Japan Plants After Suspected Cyberattack. https://threatpost.com/toyota-to-close-japan-plants-after-suspected-cyberattack/178686/

[3] 2021. Ransomware Disrupts Meat Plants in Latest Attack on Critical U.S. Business. https://www.nytimes.com/2021/06/01/business/meat-plant-cyberattack-jbs.html?smid=url-share

[4] Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. 2020. Control behavior integrity for distributed cyber-physical systems. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 30–40.

[5] Rockwell Automation Allen-Breadley. 2007. *Logix5000 Controllers General Instractions–Reference manual*. Technical Report. Publication 1756-RM003I-EN-P.

[6] Rajeev Alur and Thomas A. Henzinger. 1994. A Really Temporal Logic. *J. ACM* 41, 1 (Jan. 1994).

[7] Wissam Aoudi, Mikel Iturbe, and Magnus Almgren. 2018. Truth will out: Departure-based process-level detection of stealthy attacks on control systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 817–831.

[8] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.

[9] Francesco Basile, Pasquale Chiacchio, and Jolanda Coppola. 2016. A novel model repair approach of timed discrete-event systems with anomalies. *IEEE Transactions on Automation Science and Engineering* 13, 4 (2016), 1541–1556.

[10] F Basile, P Chiacchio, and J Coppola. 2017. An incremental model repair approach to timed discrete event systems. *IFAC-PapersOnLine* 50, 1 (2017), 13636–13641.

[11] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. 2015. Regression Verification for Programmable Logic Controller Software. In *Formal Methods and Software Engineering*.

[12] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A unifying view on SMT-based software verification. *Journal of automated reasoning* 60, 3 (2018), 299–335.

[13] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. 2012. Arcade.PLC: A Verification Platform for Programmable Logic Controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*.

[14] Henrik Carlsson, Bo Svensson, Fredrik Danielsson, and Bengt Lennartson. 2012. Methods for reliable simulation-based PLC code verification. *IEEE Transactions on Industrial Informatics* 8, 2 (2012), 267–278.

[15] John H Castellanos, Martin Ochoa, Alvaro A Cardenas, Owen Arden, and Jianying Zhou. 2021. AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 235–250.

[16] Y. Chen, C. M. Poskitt, and J. Sun. 2018. 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)*.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[18] Jasmin Dzinic and Charlie Yao. 2013. *Simulation-based Verification of PLC Programs Master of Science Thesis in Production Engineering*. Master's thesis. Chalmers University of Technology, Sweden.

[19] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2021. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[20] fischertechnik 2021. fishcertechnik.de – Building blocks for life. https://www.fischertechnik.de

[21] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit.. In *NDSS*.

[22] Luis Garcia, Stefan Mitsch, and André Platzer. 2019. HyPLC: Hybrid programmable logic controller program translation for verification. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. 47–56.

[23] L. Garcia, S. Zonouz, Dong Wei, and L. P. de Aguiar. 2016. Detecting PLC control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*.

[24] Luis Garcia, Saman Zonouz, Dong Wei, and Leandro Pfleger De Aguiar. 2016. Detecting PLC control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*. IEEE, 67–72.

[25] Shengjian Guo, Meng Wu, and Chao Wang. 2017. Symbolic Execution of Programmable Logic Controller Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

[26] Muluken Hailesellasie and Syed Rafay Hasan. 2018. Intrusion Detection in PLC-Based Industrial Control Systems Using Formal Verification Approach in Conjunction with Graphs. *Journal of Hardware and Systems Security* 2, 1 (2018), 1–14.

[27] Hannes Holm, Martin Karresand, Arne Vidström, and Erik Westring. 2015. A survey of industrial control system testbeds. In *Nordic Conference on Secure IT Systems*. Springer, 11–26.

[28] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. 2019. UChecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 581–592.

[29] Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. 2015. Runtime-Monitoring for Industrial Control Systems. *Electronics* 4, 4 (dec 2015), 995–1017.

[30] William Knowles, Daniel Prince, David Hutchison, Jules Ferdinand Pagna Disso, and Kevin Jones. 2015. A survey of cyber security management in industrial control systems. *International journal of critical infrastructure protection* 9 (2015), 52–80.

[31] Martin Kölbl, Stefan Leue, and Thomas Wies. 2019. Clock Bound Repair for Timed Systems. In *Computer Aided Verification*.

[32] Martin Kölbl, Stefan Leue, and Thomas Wies. 2019. Clock bound repair for timed systems. In *International Conference on Computer Aided Verification*. Springer, 79–96.

[33] Martin Kölbl, Stefan Leue, and Thomas Wies. 2020. TarTar: A Timed Automata Repair Tool. In *International Conference on Computer Aided Verification*. Springer, 529–540.

[34] I. Kovalenko, M. Saez, K. Barton, and D. Tilbury. 2017. SMART: A System-Level Manufacturing and Automation Research Testbed. *Smart and Sustainable Manufacturing Systems* 1, 1 (2017), 232–261.

[35] Robert Lee, Michael Assante, and Tim Conway. 2021. 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.

[36] R. M. Lee, M. J. Assante, and Tim Conway. 2021. German Steel Mill Cyber Attack. https://ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks_Facility.pdf.

[37] Yao Liu, Peng Ning, and Michael K. Reiter. 2009. False Data Injection Attacks against State Estimation in Electric Power Grids. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*.

[38] Efrén López-Morales, Carlos Rubio-Medrano, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang, Tiffany Bao, and Gail-Joon Ahn. 2020. HoneyPLC: A Next-Generation Honeypot for Industrial Control Systems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 279–291.

[39] Stephen McLaughlin and Patrick McDaniel. 2012. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*.

[40] Stephen McLaughlin, Saman Zonouz, Devin Pohly, and Patrick McDaniel. 2014. A Trusted Safety Verifier for Process Controller Code. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*.

[41] S Mesli-Kesraoui, A Toguyeni, A Bignon, F Oquendo, D Kesraoui, and P Berruet. 2016. Formal and joint verification of control programs and supervision interfaces for socio-technical systems components. *IFAC-PapersOnLine* 49, 19 (2016), 426–431.

[42] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. 2015. A CEGAR Tool for the Reachability Analysis of PLC-Controlled Plants Using Hybrid Automata. In *Formalisms for Reuse and Systems Integration*.

[43] Johanna Nellen, Kai Driessen, Martin Neuhäu$\beta$er, Erika Ábrahám, and Benedikt Wolters. 2016. Two CEGAR-based Approaches for the Safety Verification of PLC-controlled Plants. *Information Systems Frontiers* 18, 5 (Oct. 2016), 927–952.

[44] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 252–269.

[45] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral Resource-aware Model Inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*.

[46] Blake C Rawlings, John M Wassick, and B Erik Ydstie. 2018. Application of formal verification and falsification to large-scale chemical plant automation systems. *Computers & Chemical Engineering* 114 (2018), 211–220.

[47] Frances Robles and Nicole Perlroth. 2021. 'Dangerous Stuff': Hackers Tried to Poison Water Supply of Florida Town. https://www.nytimes.com/2021/02/08/us/oldsmar-florida-water-supply-hack.html.

[48] Esha Sarkar, Hadjer Benkraouda, and Michail Maniatakos. 2020. I came, I saw, I hacked: Automated Generation of Process-independent Attacks for Industrial Control Systems. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 744–758.

[49] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. 2018. Denial of engineering operations attacks in industrial control systems. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 319–329.

[50] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.
[51] Ruimin Sun, Alejandro Mera, Long Lu, and David Choffnes. 2020. SoK: Attacks on Industrial Control Logic and Formal Verification-Based Defenses. *arXiv preprint arXiv:2006.04806* (2020).
[52] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1232–1243.
[53] David I Urbina, Jairo A Giraldo, Alvaro A Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1092–1105.
[54] Zeyu Yang, Liang He, Peng Cheng, Jiming Chen, David KY Yau, and Linkang Du. 2020. PLC-Sleuth: Detecting and Localizing {PLC} Intrusions Using Control Invariants. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 333–348.
[55] Hyunguk Yoo and Irfan Ahmed. 2019. Control logic injection attacks on industrial control systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 33–48.
[56] 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. 2019. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland'19)*.
[57] Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. 2021. A Systematic Framework to Identify Violations of Scenario-dependent Driving Rules in Autonomous Vehicle Software. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 2 (2021), 1–25.

# A PARSING PLC CODE

In this section, we introduce how we translate PLC code to general-purpose high-level programming languages such as Python for modeling PLC execution. We use the translation from Ladder Logic to Python as a demonstration but the methodology is general for other PLC code languages such as Structured Text (ST), Instruction List (IL), Sequential Function Charts (SFC), and Function Block Diagram (FBD).

Due to the nature of PLC execution, all PLC code languages have similar properties. First, the code is executed in scan cycles. In each cycle, the PLC reads sensor signals into an *input memory*, executes the code based on the input signals and the current PLC state, and finally writes output signals to an *output memory*. As a result, there are three types of variables in PLC code: *input variables* read from the *input memory*, *state variables* representing the state of the PLC, and *output variables* written to the *output memory*. Second, the PLC logic is event-driven. The tasks in ICSs are mostly event-driven, as mentioned in §5. For instance, the robot arm picks the part when the sensor observes the existence of the part, the stop releases the pallet when the PLC acknowledges part processing is done, *etc.* To implement such event-driven behavior, the PLC code can be regarded as a set of if-then blocks: if the input variables and state variables satisfy a specific pattern, the PLC modifies state variables and output variables.

Given the properties of PLC code, the translation takes three main steps, similar to *TSV* [40] and *SCADMAN* [4].

First, for each PLC, we identify the entry and exit of its PLC code and distinguish three types of variables. For Ladder Logic, the entry of execution is a special routine labeled as "main routine" (routines in Ladder Logic are similar to functions in Python). The variable types are also well annotated in the source code of Ladder Logic.

Second, we do the instruction-level translation. Instructions of Ladder Logic are called Rung instructions. One Rung instruction contains a set of operators that are executed in a sequence, which defines a control flow. The control flow can be translated to Python's if-then blocks. For instance, `EQU` operator examines whether two variables are equal in value and only if two variables are equal, the next operator can be executed. In this case, we can place the "equal to" logic in Python's if condition. For advanced PLC instructions such as TON (Timer ON), we implement the equivalent python-version functions following the official documentation of PLC programming [5]. For PLC-specific data structures, *e.g.,* timers and counters, we follow the approach proposed in *TSV*[40]. One example is shown as follows.

**Ladder Logic**:
```
EQU(UpdateStep_RFID.in,15)          // Equal to
XIO(RFID_AtStop)                    // Examine If Open
TON(UpdateFinishedDelay,?,?);       // Timer ON
```
**Python**:
```
if UpdateStep_RFID == 15 and not RFID_AtStop:
  TON(UpdateFinishedDelay)          // python-version TON
```

Third, we combine PLC code from multiple PLCs (*e.g.,* Fischertechnik testbed has four separated PLCs) together to generate one single function block, which is the PLC execution model we need for system modeling. We adopt the method proposed in *SCADMAN*: appending all of the function and function block definitions and merging the main PLC program of each PLC.

# B MINING TEMPORAL EVENTS

We introduce steps of mining temporal events in detail.

For each testbed, we first collect 10-hour data traces of the PLCs, including both successful and failed execution of the ICS under various configuration settings. Each data trace records the values of PLC variables on each scan cycle during the execution of the ICS testbed. Formally, we can define the data trace as an ordered list $DT = \{(t_1, s_1), \ldots, (t_n, s_n)\}$ where each element is a tuple of one timestamp and a PLC state (*i.e.,* values of PLC variables).

Second, we assign PLC variables into various groups to make the data mining more oriented. The variables should be in one group if they have causal relation and one variable can join various groups. First, PLC variables connecting with the same device are in the same group (mentioned in §5.3). When developers configuring the system devices and their interactions, PLC variables are automatically labeled to be outgoing signals (*e.g.,* commands sent to other devices) or incoming signals (*i.e.,* sensor signals), associated with specific device IDs. By definition, the outgoing and incoming signals are likely to have causal relations and thus may contain useful temporal invariants. For example, the time interval between the outgoing commands to a robot and its incoming signals may indicate the time for the robot to accomplish specific tasks. Second, some variables have causal relations because of the physical environment. For instance, sensor signals for detecting parts, the conveyor speed, the state of stop gates, *etc.* are all about the position of the part. We require the developer's annotations to put them into one group so that the data mining can reveal the invariants about physical dynamics more efficiently. The grouping is an enhancement on VetPLC's method to improve the accuracy and efficiency of mining temporal invariants.

Third, we transform data traces to sequences of state updates. One state update is one modification on the PLC states. For each data trace and for each variable group, we compare two PLC states in every two consecutive scan cycles and then record changed

**Algorithm 2:** Safety constraint generation.

**Input:** SESG $G = (V, E, \alpha, \beta, \gamma)$.
**Output:** Safety constraint $C_{safe}$.

$C_{safe} \leftarrow 1$;
**for** $\pi \in$ *paths in* $G$ **do**
  **if** SmtCheck($C_p^\pi \wedge C_s^\pi \wedge C_{safe}$) = *sat* **then**
    $C_{safe}^\pi \leftarrow \forall T^\pi (C_p^\pi \implies C_s^\pi)$;
    $C_{safe} \leftarrow C_{safe} \wedge$ QuantifierElimination($C_{safe}^\pi$);
  **end**
**end**

variables, their previous values, and their new values as one state updates. In order words, one state update is a triple of modified variables, previous values, and new values, formally $(V, x, y)$. By iterating all scan cycles in the data trace, a sequence of state updates is generated: $US = \{(t_1, V_1, x_1, y_1), \ldots, (t_k, V_k, x_k, y_k)\}$.

Fourth, we use Perfume [45] algorithm to discover the temporal invariants in the sequences of state updates. For each variable group, we have a set of sequences of state updates. Perfume can directly analyze the sequence sets, regard each state update as one item, and find temporal properties among the items. Perfume supports various types of temporal properties and we are interested in two of them (the same as VetPLC): "a always followed by b upper-bound t" and "a always followed by b lower-bound t". The properties we are mining are exactly the time interval bounds of temporal events: one state update (operation) follows another state update (trigger) with a bounded interval range. Hence, each temporal invariant specifies a pair of state updates and a time interval bound: $I = ((V_a, x_a, y_a), (V_b, x_b, y_b), t_{min}, t_{max})$.

Especially, for variable groups with configuration parameters, we further split the sequences of state updates into subsets and the configuration parameters have the same values in each subset. If we do the invariant mining on each subset, for each pair of state updates, we may have temporal invariants with different time interval bounds. If so, it means the configuration parameters have an impact on the temporal properties. To discover the relationship between configuration parameters and time interval bounds, we apply linear regression whose input values are configuration parameters values and output values are the tine interval bounds. Therefore, we can merge the invariants under various configuration parameters into one final invariant whose time interval bound is the result of linear regression, a function of configuration parameters.

Finally, we translate the temporal invariants back to temporal events through a one-to-one mapping. The event trigger is the result of the first state update ($V_a = y_a$) while the event operation is the second state update ($V_b \leftarrow y_b$). The time interval bounds remain the same. Formally, the temporal invariant $I = ((V_a, x_a, y_a), (V_b, x_b, y_b), t_{min}, t_{max})$ is mapped to temporal event $E = (V_a = y_a, (t_{min}, t_{max}), V_b \leftarrow y_b)$.

# C IMPLEMENTATION OF SAFETY CONSTRAINT GENERATION

Though the formula of the safety constraint (Equation 1) is straightforward, state-of-the-art SMT solvers (*e.g.,* Z3) get slow when solving complicated formulas such as quantifiers (*i.e.,* $\exists, \forall$). The implementation of the safety constraint generation needs to be carefully designed to minimize the time cost of both offline and online processes.

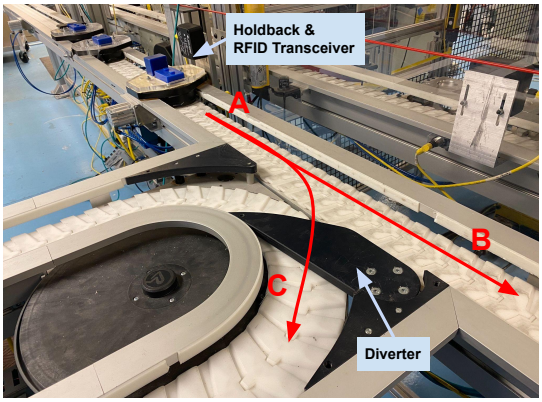



**Figure 7: A photo of SMART's diverter.**

Therefore, instead of generating the safety constraint exactly following the formula, we do two improvements in the implementation, as shown in Algorithm 2. First, SmtConf does quantifier elimination to produce a quantifier-free safety constraint, which can reduce the latency of online verification and repair. Second, violation paths that are already identified by the safety constraint are skipped. In Algorithm 2, quantifier elimination, which is an expensive process, is only done on reachable paths with violations but satisfying the safety constraint. As a result, the time cost of the offline safety constraint generation is reduced. Obviously, the generated safety constraint is equivalent with Equation 1. Quantified performance metrics are shown in §7.

# D ICS TESTBEDS

We introduced the testbed *SMART cell* in §3.1. In this section, we illustrate the behavior of the other two ICSs, which we also evaluate SmtConf on: *SMART diverter* and *Fischertechnik*.

## D.1 SMART Diverter

The diverter is one critical component in testbed SMART, which ensures the pallets move along correct routes. In this scenario, as shown in Figure 7, there is one main straight conveyor and another branch conveyor. Pallets coming in through entry (A) may leave this component through the exit of either the main conveyor (B) or the branch conveyor (C). The route of pallets depends on the RFID of parts on them and the RFID indicates whether the part has been finished. Finished parts should leave the system through the branch conveyor while unfinished parts should remain on the main conveyor to be further processed. The diverter controls the routes of pallets.

**Workflow.** SMART takes the following steps to route pallets to their right destination. (1) A pallet arrives and stops at the closed holdback gate; (2) one RFID transceiver deployed at the holdback reads the RFID of the pallet; (3) if the RFID is correctly read and the pallet has been waiting at the holdback for long enough (implemented using a timer), set the status of the diverter according to read RFID and meanwhile open the holdback; retract the diverter if the pallet is holding an unfinished part and vice versa; the pallet will pass the holdback; (4) close the holdback again after a short time; (5) eventually, the pallet passes the diverter and goes to its destination determined by the diverter.

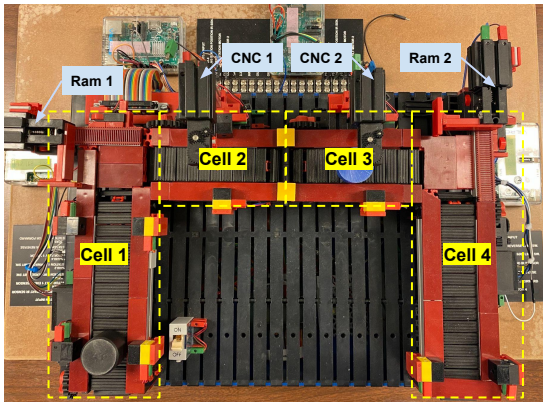
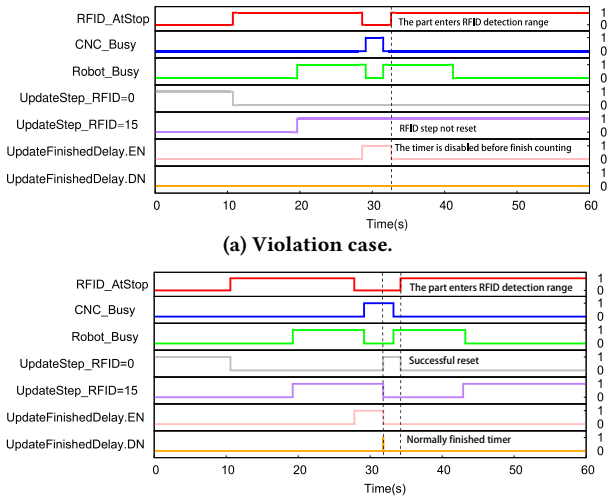



**Figure 8: A photo of manufacturing sector testbed based on Fischertechnik.**




**(a) Violation case.**




**(b) Repaired case.**

**Figure 9: Signal values during the motivating example produced by simulation.**

**Case study of case #10** (Table 4). The basic specification for SMART diverter is its correct routing functionality. Though the above workflow works well for processing one pallet, violations may occur when there are multiple pallets arriving at the holdback consecutively. Considering a scenario when two pallets (*i.e.,* pallet #1 and pallet #2) arrive at the holdback with a time interval of 8 seconds, the SMART diverter works correctly because when pallet #2 is released by the holdback, pallet #1 has already passed the diverter, and in this situation, both pallets go to their right destination. However, when the pallet interval is tuned shorter (*i.e.,* 6 seconds), pallet #1 has not passed the diverter when pallet #2 is released by the holdback. If the status of parts is different on two pallets, the release of pallet #2 causes a change in the status of the diverter, which makes pallet #2 go to the wrong destination. In summary, because of the absence of a sanity check about whether the conveyor between the holdback and the diverter is clear, SMART relies on a long enough pallet interval to avoid conflicts between two consecutive pallets. Unfortunately, improperly short pallet intervals are possible to be injected into the system.

In case #10, SmtConf generates the SESG representing system behavior when handling two consecutive pallets with a symbolic pallet interval. SmtConf's verification reveals the violation under a short pallet interval of 6 seconds (40 violation paths out of 54 possible paths in SESG) and SmtConf's repair generates the minimal safe pallet interval as 6.76 seconds (the ground truth is 6.6 seconds). The verification takes only 0.04 seconds so that the improper configuration is reverted before causing any wrong routing results. The recommendation of proper pallet intervals also helps developers to realize the highest production while avoiding safety problems.

## D.2 Fischertechnik Testbed

This testbed is a manufacturing system that contains four cells (Cell 1 to Cell 4), as shown in Figure 8. Two CNCs (CNC 1 and CNC 2) are located at Cell 2 and Cell 3 respectively. Each cell has one conveyor belt and one or two sensors detecting the presence of parts. Two rams (Ram 1 and Ram 2) are deployed to push parts from Cell 1 to Cell 2 and from Cell 3 to Cell 4 respectively. All the above devices are controlled by one PLC. 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 via Ethernet and linked via Modbus.

**Workflow.** A successful task execution of Fischertechnik means the system accepts one part at the entry of Cell 1, processes the part in CNC1 and CNC2, and finally delivers the part to the exit of Cell 4. During the whole process, the part arrives Cell 1, Ram 1, Cell 2, CNC 1, Cell 3, CNC 2, Ram 2, and Cell 4 in order. Especially, the transition of parts from Cell 1 to Cell 2 takes the following steps. (1) The part arrives at one sensor on Cell 1, which triggers the counting of one-timer called *TransitingToRam*; (2) the part arrives the retracted Ram 1 through the conveyor belt; (3) timer *TransitingToRam* reaches one threshold (by default 2 seconds) and PLC sends Ram 1 a signal to release Ram 1 and push the part to Cell 2.

**Case study of case #14** (Table 4). This case focuses on the transition of parts from Cell 1 to Cell 2 and the specification requires that rams are retracted when the part arrives (*i.e.,* no collision between the part and rams). When the threshold of timer *TransitingToRam* is 2 seconds, since the conveyor needs less than 1 second to deliver the part from Cell 1's sensor to Ram1, step (3) always happens after step (2) which is normal behavior. However, if we use a shorter timer threshold of 0.9 seconds to increase overall system production, step (3) may happen too early that the part has not arrived at Ram 1, which is a safety violation.

In case #14, when the threshold is improperly set to 0.9 seconds, SmtConf's verification discovers 6 violation paths from 8 possible paths in total in SESG and the violation detection takes only 0.35 seconds. SmtConf automatically generates the minimal safety threshold of 0.34, which is exactly the maximal time the conveyor takes to deliver the part from Cell 1's sensor to Ram 1. SmtConf only can eliminate the safety violation in less than one second but also provide a better configuration choice to maximize system production.

# E SIMULATION

It is inefficient and risky to reproduce all safety violations on real physical devices of ICSs. As a result, we use simulators instead to study the system behavior under various configuration parameters.

**Implementation.** The simulator contains three main components: (1) a PLC executor which runs PLC code and maintains the status of PLC's memory; (2) a handler of physical dynamics which predicts physical properties of all objects in the ICS scenario, *e.g.,* position, and velocity of parts and pallets; and (3) a log management system recording all runtime information of the ICS execution. The simulation maintains a clock and is executed in cycles. In each cycle, the simulator executes the PLC code to update PLC's memory, update physical properties of all in-system objects, and increase the clock value by a fixed small time interval (10 ms in our implementation). Similar to the real ICSs, the configuration parameters can be configured by users.

**Example simulation results.** We show the simulation of the motivating example (test case #4 in Table 4) before and after the repair in Figure 9a and Figure 9b respectively. In Figure 9a, the CNC's motor frequency is 5,000 rpm and there is a violation against specification "skipped RFID update" (#2 in Table 2). When the part enters the RFID detection range (signal `RFID_AtStop` goes positive), the timer `UpdateStep_RFID` has not finished its counting behavior and the `UpdateStep_RFID` is not reset. In Figure 9b, the motor frequency is repaired to 3,814 rpm so that the RFID update becomes normal (`UpdateStep_RFID=0` happens before `RFID_AtStop=1`). The above simulation results are consistent with the data traces collected from the real-world execution of SMART.