Dynamically Detecting Invariants for Automatic Testing PLC Programs

Zeyu Lu[†], Xia Mao[†], Yanhong Huang[†§*], Jianqi Shi[†‡], Yang Yang[†]
[†]National Trusted Embedded Software Engineering Technology Research Center
East China Normal University, Shanghai, China
[‡]Hardware/software Co-Design Technology and Application Engineering Research Center, Shanghai, China
[§]Shanghai Key Laboratory of Trustworthy Computing, Shanghai, China
Email: {zeyu.lu, xia.mao, yang.yang}@ntesec.ecnu.edu.cn, {yhhuang, jqshi}@sei.ecnu.edu.cn

***Abstract*—Since programmable logic controllers (PLCs) control safety-critical infrastructures, examining the PLC software satisfies the high-reliability specifications necessary to ensure the safeness of PLCs. However, prior works have limitations in finding defects in the PLC source code. Static verification techniques suffer from notable false positives without capturing runtime behavior. The symbolic execution and conformance testing technique captures the relations of inputs and outputs. It is not sufficient to consider only the data constraints as the PLC operates in real-time. In this paper, we propose a novel approach in the detection of the runtime behavior of PLC programs with incorporated time constraints. This testing approach automatically finds implementation errors in PLC programs by mining invariants from runtime traces. As the existing tools mine only data or time invariants which are inadequate to test PLC programs, our approach focuses on the interplay of data and time invariants. Dynamically detected data-time invariants are then checked with the safety specifications. We evaluate the usefulness of our approach in a real-life case. The experimental results show that the proposed approach can find errors in PLC programs effectively.**

***Index Terms*—programmable logic controller, program invariant, real-time system, interplay**

I. Introduction

Industrial Control Systems (ICS) have been used widely in many safety-critical domains, such as smart power grids, nuclear power plants, and transportation systems. These areas play an essential role in modern society. A programmable logic controller (PLC) is an industrial computer that is capable of being programmed to perform control functions. Now, PLCs are the most widely-used industrial process control technology. Since the PLC software controls safety-critical infrastructures, its inherent defects may have severe consequences, such as financial and property losses.

To ensure control logic safety, many previous studies [1]–[4] statically verified PLC programs to discover security bugs. These studies converted the PLC program to a model checker's input language, such as NuSMV [5] or UPPAAL [6], and the model checker automatically checked whether the program satisfied the given formal specification. However, these approaches suffer from notable false positives because the error checking was performed statically without the program having been executed. These approaches found violation paths that could not be executed at runtime. Besides, these approaches are at the abstract model level, which is more suitable for checking design defects rather than implementation errors.

In recent years, several studies [7], [8] used symbolic execution and concolic testing to automatically generate test cases for PLC programs. In addition, Provost et al. [9] used conformance testing to test whether the execution code of the PLC conforms to the specification. Although these studies can perform white box testing by using symbolic execution or black-box testing based on the conformance testing technique, these techniques focus only on the relationship of inputs and outputs. It is sufficient to capture data value relations for typical software without considering the timing of the system under test. However, it is inadequate for the testing of time-constrained software since PLC is a real-time system.

To address the difficulties in detecting the runtime behavior of PLC programs with incorporated time constraints, we propose a novel approach to test PLC programs. Our approach is at the code level. It mines program invariants from runtime traces of the program under test. A program invariant, or property, is a condition that holds at a given point. Mining invariants from runtime traces eases the notable spurious warnings result from the static analyzers. Our technique further mines time invariants, considering PLCs operate in real-time.

However, existing dynamic invariant detection tools extract one-dimensional models, such as data or time, without capturing the interplay of them. Both data and time models are useful in that PLCs to operate in real-time and the data values of variables also express events that occur in the system. In this paper, we focus on the interplay of data and time invariants to find source code defects in PLC programs. Different program invariants capture different runtime behaviors during execution. *Data Invariant* expresses the range of values assigned to the variables and the relation of values of different variables. *Time Invariant* describes the time boundary of events that occur in a system. *Data-Time Invariant* illustrates the timing constraints of data invariants.

Our automatic testing approach is threefold. First, we instrument the program under test with the input and output relations. Second, we mine data-time invariants to observe the time performance of the implemented control logic. Third,

*Corresponding Author
DOI reference number: 10.18293/SEKE2021-105

the dynamically detected data-time invariants are compared with the manually crafted specification, which expresses the expected behavior of the PLC software. Besides, we mutate the existing test suites to obtain adequate test cases to improve the quality of invariants derived. Once the PLC program has been tested and satisfies the safety requirements, it will be downloaded to real PLCs.

We evaluate the effectiveness of our proposed approach on a real-life case, i.e., cosmetic packing process. The experimental results show that the dynamically discovered invariants are efficient to help test PLC programs. We have found a latent error that is not easy to discover by using the existing approach such as symbolic execution in our implementation of the cosmetic packing system. *To the best of our knowledge, we are the first to mine data-time invariants of programs dynamically in the context of ICS*. In summary, this paper makes the following contributions:

1) We propose a testing technique that uses dynamically detected invariants to discover implementation errors in PLC programs at the code level.

2) We propose methods to derive data-time invariants from execution traces of PLC programs. The data-time invariants express the runtime behavior of the PLC programs more accurately.

3) We perform static analysis of the program under test to derive data invariants specifically tailored to PLC programs.

The rest of the paper is organized as follows. Section II provides some background and a motivational example. Section III illustrates our approach. Section IV presents the evaluation results of our approach. Section V discusses related work. Section VI concludes the paper.

II. Preliminary and Motivation

A. Programmable Logic Controller

The program of PLC is executed continuously and each execution is called a *scan cycle*. Each scan cycle of a PLC consists of the following three processes: (1) sensor measurements are read to input variables, (2) the control commands are computed based on sensor values and the control logic, and (3) the control commands are sent to actuators which change the physical processes. The PLC sits in the closed-loop to perform control functions.

PLC Programming Languages. IEC 61131-3 standard is the third part of the IEC 61131 standard which provides standards to programmable controllers. There are five programming languages included in the IEC 61131-3 standard: namely, ST, IL, LD, FBD, and SFC. The five programming languages share many common elements and can be transformed with each other [10]. In this paper, we focus on the *Structured Text* (ST) language which offers a flexible way of expressing complex functionality.

B. Motivation Example

We present an example of a flashing light to illustrate that the runtime properties detected by existing tools cannot accurately reflect the behavior of a program. It is essential to mine the combined data and time properties of PLC programs to deal explicitly with time measures.

```
1  IF Weight >= MaxValue THEN
2      Overdrive := TRUE;
3  ELSE
4      Overdrive := FALSE;
5  END_IF;
6  Flash_1(StartFlash := Overdrive);
7  PL1 := Flash_1.FlashOut;
8  Timer01(IN:= Start, PT := T#1s);
9  Motor := Timer01.Q;
10 Solenoid := OnOff AND NOT Overdrive;
```

Fig. 1. Code Snippet of a Flashing Light Program

The expected behavior of the PLC program is as follows: if the weight on the conveyor exceeds a pre-defined constant value for some time, a warning light PL1 will begin to flash. Additionally, the solenoid is de-energized. The light PL1 flashes with the time interval of one second. If there is no anomalous situation, the PL1 light will keep off state. If the start button is pressed, the motor will start one second later.

The snippet of the implemented ST program is shown in Figure 1. The program has three input variables, i.e., *Weight*, *OnOff*, and *Start*, and three output variables, i.e., *PL1*, *Motor*, and *Solenoid*. The output variable PL1 is connected to a lamp. The state of PL1 can be judged from whether the lamp lights up (line 7). The delayed start of the motor is controlled by a timer (lines 8-9). The state of Solenoid is affected by whether the weight exceeds the preset value or not (line 10).

To test whether the implemented program satisfies the expected behavior, we first use Daikon [11] to detect runtime invariants. After running Daikon, the derived data invariant of variable PL1 is *"Weight >= MaxValue" ==> PL1 one of* {*false, true*}. Besides, if we specify the *MaxValue* as 26 and use the approach in [12] to find *Weight* equals 26 in the execution trace manually, Perfume [13] will infer the invariant *Weight=26 → PL1 [1s, 2s]*. However, Daikon can only detect invariants of data value relations. It cannot express the time boundary of the relations. The invariant mined only by Perfume cannot describe the relationship of predicate *Weight >= MaxValue* with variable *PL1*.

In this example, the derived data-time invariant by applying our approach is *Weight >= MaxValue → PL1 [1s, 2s]* when the scan cycle of the program is 50 milliseconds. This invariant expresses that, if the weight exceeds a preset value, the light PL1 will stay off for at least 1 second and will then turn to on for 1 second. Besides, it shows that the runtime behavior is correct in the existence of timers, which delays the operation for one second.

III. Approach

A. Overview

Figure 2 shows the core workflow of our approach. The process of the test workflow can be summarized as follows:

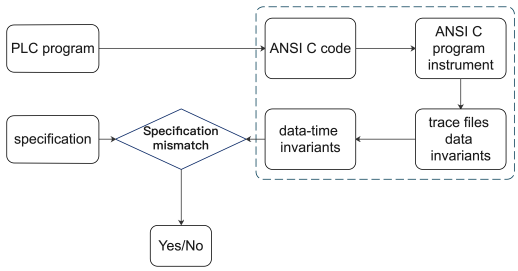

Fig. 2. Automated Test Workflow for PLC Programs

We first use an open-source PLC compiler named matiec [14] to compile the PLC program to ANSI C code. The C code is semantically equivalent to PLC programs, and there are existing works [7], [12] using the matiec compiler to test PLC programs. For the translated C program, we instrument it using static analysis to leverage Daikon mining data invariants. Daikon's front-end produces a trace file that records the values of the variables, and Daikon dynamically detects data invariants based on the trace file. After data invariants have been derived, we combine Perfume to capture the time constraints of data invariants. The detected data-time invariants are checked with the manually crafted specifications. The mismatches between the detected invariants and the specifications indicate that the PLC program contains errors. In addition, to improve the quality of detected invariants, we mutate test cases generated by symbolic execution to produce adequate test cases.

B. Instrument ANSI C Programs

To leverage Daikon to detect invariants tailored to PLC programs, we instrument C code with additional program points, i.e., instrument code with dummy procedures [11]. The dummy procedures do not affect the normal execution of the program and can help Daikon detect invariants concerning specific variables. The arguments of a dummy procedure include the variables of which we want Daikon to detect invariants and the timestamp that records the calling time of the dummy procedure.

For C or Java program, Daikon infers invariants at the granularity of function. However, the PLC program is not composed of functions. There are three types of program organization units (POUs) in PLC, i.e., *Program*, *Function Block*, and *Function*. Each PLC program may be comprised of several POUs. If the invariants are detected at the level of POU, the granularity is too coarse as there may be hundreds of lines of code in one POU. By contrast, if the invariants are generated at the statement granularity, then the number of detected invariants is huge and some of them do not make sense. We detect the invariants at the granularity of several correlated statements. The granularity is coarser compared to the statement level and finer compared to the POU level. These statements start with the one that input variables lie in and end with the one that the output variable sits. The change of the inputs impacts the output. We perform static analysis to obtain all input-output relations and instrument the ANSI C code with these relations.

We first generate the control-flow graph (CFG) of the PLC program. Every statement in the program is represented as a node in the CFG. The *control-dependence* and *data-dependence* are computed from the CFG. We then obtain the program dependence graph (PDG) based on the computed control and data dependence. We start from each node containing input variables and then traverse the PDG by depth-first search (DFS). If the traverse reaches a node that contains the definition of an output variable, the statements both in the node and the start node will be output. The traversal will output all of the input-output node pairs in the form of *"if-then"* relations. The statement in the input node is the condition and the output variable is the return variable. We pass the variables in the two nodes as arguments to the dummy procedure and add the *"if-then"* relations in the dummy procedure body. For example, the dummy procedure body of the two nodes *"Weight >= MaxValue"* and *"PL1:= Flash_1.FlashOut"* in the motivation example is:

if (Weight >= MaxValue)
return PL1;
return PL1;

Specifically, there exists a case where a node contains both input and output variables. In this case, we simply pass the output variable in the node as arguments to the dummy procedure.

C. Invariant Types

Perfume mines property types based on the execution log and formalizes the mined properties using timed propositional temporal logic (TPTL) [15]. TPTL is a real-time specification language for the specification of real-time systems.

By default, Perfume mines seven property types, and we focus on four of them:

$\Box x.(p \rightarrow (\Diamond y.(q \wedge y - x \leqslant t)))$ / $\Box x.(p \rightarrow (\Diamond y.(q \wedge y - x \geqslant t)))$ whenever there is p in a trace, p is followed by q in the trace at time y with a time difference of at most/least t.

$\Box x.(p \text{ U } (\Diamond y.(q \wedge y - x \leqslant t)))$ / $\Box x.(p \text{ U } (\Diamond y.(q \wedge y - x \geqslant t)))$ whenever there is q in a trace, p is preceded by q in the trace at time y with a time difference of at most/least t.

For the motivation example in Figure 1, the TPTL formula of the detected invariant is

$\Box x.$*(Weight >= MaxValue* $\rightarrow$ *(*$\Diamond y.$*(PL1* $\wedge$ *y - x* $\geqslant$ *1s)))*.

In this paper, we use shorthand notations to represent the TPTL formulae. For the first two TPTL formulae, the notation is p $\rightarrow$ q [t_{min}, t_{max}]. Similarly, the notation p U q [t_{min}, t_{max}] corresponds to the last two formulae.

Besides the four TPTL property types above, we also leverage Perfume to derive other types of invariants as presented in Table I.

Type 1. The first invariant represents a variable that is assigned a certain value and kept for the duration of t_{max}. The second and the third invariants express that under the predicate *expr*, the output holds for the duration of t_{max}.

TABLE I
TYPE OF DERIVED DATA-TIME INVARIANTS

Type	Data-Time Invariant
1	var = [] t_{max} expr → var = [] t_{max} expr U var = [] t_{max}
2	var = [] t_{min}, t_{max}
3	var = [] → var = [] t_{min}, t_{max} var = [] U var = [] t_{min}, t_{max}
4	expr → var = [] t_{min}, t_{max} expr U var = [] t_{min}, t_{max}

Type 2. This invariant denotes the time interval that a variable takes a specific value.

Type 3. The two invariants describe a variable that is assigned different values with the time bounded by t_{min} and t_{max}.

Type 4. These two invariants express the relationship between the condition *expr* and the corresponding variable assignments in the time difference t_{min} and t_{max}.

D. Generate Data-Time Invariants

The invariants dynamically detected by Daikon provide the relations of values of variables. After data invariants are generated, we process the trace file produced by Daikon's front-end and apply Perfume to derive data-time invariants.

Trace Process. For each data invariant, we first extract all related data-trace records from the trace file. Each data-trace record includes runtime value information in one scan cycle. For the data invariant *"Weight >= MaxValue" ==> PL1 one of* {*false, true*} in the motivation example, we only extract the data-trace records corresponding to the predicate *Weight >= MaxValue* and the variable *PL1* from the trace to detect the time boundary of the data invariant.

Given that the trace only contains the extracted data-trace records, we convert each record to a tuple p_k = [(*name1=value1*, *name2=value2*, ⋯, *timestamp*), *(name=value, timestamp)*], where *name1=value1*, *name2=value2*, ⋯, is the variables' names and values in predicate *expr*, *name=value* is the name and value of the output variable *var*. For convenience, we denote tuple p_k as p_k = [*expr*, *var*].

A tuple p_k contains the values of the variables contained in both *expr* and *var* in one scan cycle. The trace τ comprised of tuples p_1, p_2, ⋯, p_k, ⋯, represents all possible values that the variables in both *expr* and *var* obtain in one execution. *expr* and *var* will be evaluated true when particular combinations of input conditions are met.

Deriving Invariants. We propose four methods to derive different types of data-time invariants. If the predicate *expr* evaluates true, we replace *name1=value1*, *name2=value2*, ⋯, with the predicate. For instance, the tuple [(*Weight=27*, *MaxValue=26*, *timestamp*), (*PL1=true*, *timestamp*)] is replaced with [(*Weight >= MaxValue*, *timestamp*), (*PL1=true*, *timestamp*)].

Method 1: We extract sub-trace s_n which is comprised of the tuples p_1, p_2, ⋯, p_k that the output variable *var* keeps a certain value and the predicate *expr* evaluates true. We pass all of the sub-traces s_1, s_2, ⋯, s_n to Perfume to obtain the maximum time that the causal relation *expr -> var* holds.

Method 2: We extract sub-trace s_n which is comprised of the tuples p_1, p_2, ⋯, p_k that the output variable *var* keeps a certain value and the predicate *expr* evaluates true. For the two consecutive sub-traces (s_1, s_2), we keep the last tuple p_k in s_1 and the first tuple p_1 in s_2. Similarly, for the two consecutive sub-traces (s_2, s_3), we keep the last tuple p_k in s_2 and the first tuple p_1 in s_3. The remaining sub-traces are processed so on and so forth. For the pairs of tuples ($s_1.p_k$, $s_2.p_1$), ($s_2.p_k$, $s_3.p_1$), etc., we pass them to Perfume and obtain the time interval that the causal relation *expr -> var* holds.

Method 3: We process the whole trace and only keep the tuples that represent state transitions. Formally, for the two consecutive tuples p_{i-1}=($expr_{i-1}$, var_{i-1}) and p_i=($expr_i$, var_i), if one of the values changes in p_i compared to p_{i-1}, then tuple p_i keeps; otherwise, it will be removed. After the process finishes, we extract sub-trace s_n which is comprised of the tuples *p*=[(*expr*, *var*)] and *p'*=[(*expr'*, *var'*)] that the *expr* evaluates true and *var'* represents the occurrence of output event. In addition, *expr'* does not necessarily evaluate true and *var* does not necessarily represent the occurrence of output event. We pass all of the sub-traces s_1, s_2, ⋯, s_n to Perfume to mine data-time invariants. The time boundary mined by this method corresponds to the time of state transition.

Method 4: We extract the sub-trace (s_n, t_n) which is comprised of the tuples p_s=[(*expr*, *var*)] and p_t=[(*expr'*, *var'*)] that the *expr* evaluates true and *var'* represents the occurrence of output event. In addition, *expr'* does not necessarily evaluate true and *var* does not necessarily represent the occurrence of output event. We exclude all *var* from sub-trace s_n and exclude all *expr'* from sub-trace t_n.

1) The minimum time boundary t_{min} between *expr* and *var*: for sub-trace (s_n, t_n), we only keep the first tuple in s_n. The remained tuples in (s_n, t_n) are ($s_n.p_1$, $t_n.p_1$, ⋯, $t_n.p_k$). We pass all of the sub-traces (s_1, t_1), (s_2, t_2), ⋯, (s_n, t_n) to Perfume to mine data-time invariants.

2) The maximum time boundary t_{max} between *expr* and *var*: for sub-trace (s_n, t_n), we only keep the first tuple in t_n. The remained tuples in (s_n, t_n) are ($s_n.p_1$, ⋯, $s_n.p_k$, $t_n.p_1$). We pass all of the sub-traces (s_1, t_1), (s_2, t_2), ⋯, (s_n, t_n) to Perfume to mine data-time invariants.

In particular, there exists a case where the data invariant only includes one variable. In this case, the tuple p_k = [(*name=value*, *timestamp*)], and the above methods also hold. The trace process is simpler with only one variable involved. Take the data invariant *GreenNS one of* {*false, true*} for example, we use *Method 1* to derive the first *Type 1* invariant and use *Method 2* to derive the *Type 2* invariant.

E. Specification Mismatch

The data-time invariants are more useful for describing the runtime behavior of the PLC program because they capture the

TABLE II
MUTATION OPERATORS USED IN MUTATING TEST CASES

Mutation Operator	Example
boolean constant replacement	*var=true –> var=false*
numeric constant replacement	*var = 1.0 –> var = 2.0*
unary operator insertion	*var=5 –> var=-5*

time constraints of data invariants. The dynamically detected invariants describe the behavior of the system, while the specifications express the expected behavior of a system. After deriving the data-time invariants, we check the dynamically detected invariants for errors against manually crafted specifications.

To examine whether there exists a mismatch between the observed and expected behavior of a program, for every generated data-time invariants ϕ_i corresponding to the specifications, we manually check $\sigma_i \vDash \phi_i$, where $\sigma_i \in \Sigma$ is the actual specifications of PLC programs.

F. Test Case Generation

To generate test cases, we follow the approach of $\mathrm{S_{YM}PLC}$ [7]. The test suites generated by symbolic execution guarantee the instruction coverage. However, dynamic invariant detection requires adequate test cases to improve the quality of detected invariants. In the motivation example, if the automatically generated test case for the variable *Weight* is 27, Daikon will infer data invariant *Weight == 27*. If there are various values assign to the variable *Weight* like 47, -27, 87, 21, 28, 17, 23, 57, etc., the detected data invariant will not include *Weight == 27*, which is too concrete and makes no sense.

In this paper, we mutate existing test suites to make it suitable for dynamic invariant detection. The mutation operators are shown in Table II. Once the mutated test suites are generated, we remove the test cases which do not conform to the variable's type and allowed value ranges. In addition, redundant test cases are discarded from the mutated test suites.

IV. EVALUATION

We apply our testing approach to a representative real-life case study, i.e., cosmetic packing. The production process is common to find in the automation industry. A code error has been found in the program of cosmetic packing. The case study is conducted on the Ubuntu 18.04 LTS operating system.

A. Experimental Setup

The matiec project provides the iec2c compiler which generates ANSI C code equivalent to the original PLC program. Multiple C files are generated after code translation. To produce basic test suites, we employ KLEE [16] to perform symbolic execution in the translated C code. We use Python to create mutated test suites. During the instrument step, we employ ANTLR [17] to perform static analysis of the PLC program and then instrument translated ANSI C code with the result of static analysis. We use the gcc compiler to compile them into an executable softplc file. The executable softplc can simulate the run of the PLC program without running on a real PLC. We write a Python script to process trace files generated by Daikon's front-end and apply Perfume to the processed trace file to derive data-time invariants. The source code of our implementation is available online [18].

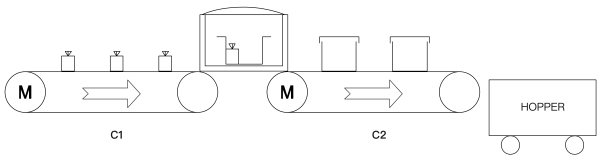

Fig. 3. Cosmetic Packing Process

TABLE III
DETECTED INVARIANTS OF COSMETIC PACKING

Specification	Data-Time Invariant	
I	reached=true	[1.5s, 3s]
	reached=false U reached=true	[1.5s, 1.5s]
II	full_box → send_box	[1.57s, 1.57s]

B. Cosmetic Packing

The cosmetic packing system packs cosmetics in a box and then puts the packed box into a hopper. Figure 3 shows the whole operation process. There are two conveyor belts C1 and C2. C1 sends cosmetics to the packing region and C2 sends the packed box to a hopper. The cosmetics move on C1 at a constant speed. Once a cosmetic reaches the packaging area, it will be put into a box. If there are three cosmetics in a box, the box will be packed one second later and C2 will transmit the box to a hopper. There are three switches *Start*, *Pause*, and *Reset*, which start, pause, and reset the process, respectively.

The specifications of the automated packing system are: I. The time interval for two consecutive cosmetics to reach the packaging area is 1.5 seconds since the packing process consumes time. II. The packed box should be present in C2 within one second after the third cosmetic is put into the box.

We execute the code in the simulator for 3000 cycles and repeats five times. Each scan cycle lasts for 50 milliseconds. There are total of 7 data-time invariants obtained, and the invariants corresponding to the specifications are listed in Table III. In Table III, the first invariant associated with Specification I represents that the two consecutive cosmetics reaches the packaging area for at least 1.5 seconds. The second invariant associated with Specification I denotes that the minimum time duration that cosmetics arrive in the packaging area is 1.5 seconds. These two invariants satisfy the specification.

An implementation error has been detected by applying our approach. In the process of cosmetic packing, it takes one second to pack the box after the third cosmetic reaches the packing area. However, for the data-time invariants derived in one simulation, the time of box packing lasts for nearly two seconds. As shown in Specification II of Table III, the

detected packing time violates the specification. Issues might occur since cosmetics repeatedly approach the packing area every 1.5 seconds.

We analyze the implemented program and simulator to find the cause of specification violations. The reason is that the timer of box packing returns to zero when we pause the packing process after the system operates for 6.5 seconds. After restarting the system, the timer starts to time from zero, which is inaccurate since the box packing process can keep its operation after restarting the system. We update the implementation and the detected invariant *full_box → send_box [1s, 1s]* satisfies the specification. Concretely, in the original implementation, we use an *on-delay timer* (TON) to time the box packing. However, TON does not retain the elapsed time if the input goes false. We switch TON to *retentive on-delay timer* (RTO) which retains the elapsed time when the pause switch is pressed.

C. Discussion

The scalability problem arises when the number of program points instrumented increases during performing dynamic invariant detection. In addition, the number of scan cycles executed also brings time and space costs. To improve scalability in the automatic testing process, one could reduce either the number of program points instrumented or the number of scan cycles. Reducing the number of cycles executed in one execution and repeating several times execution can tackle the scalability issue.

V. Related Work

Sallai et al. [19] generate x86-representations of PLC programs to test, simulate, and visualize PLC programs. They transform PLC programs into C, Scilab, and Java programs. The semantically equivalent x86 representation which can execute on personal computers overcomes the lack of advanced tools to help PLC programming. Our work converts PLC programs into C to simulate the execution of the PLC code.

PLCInspector [20] mines either linear temporal logic (LTL) specification using Texada or data invariants using Daikon from runtime traces of PLC programs. PLCInspector does not combine Texada with Daikon to detect data-temporal properties. Besides, the mined LTL specification cannot describe the runtime behavior of PLC programs properly. As PLC runs in real-time, LTL is not suitable to quantitatively express the time boundary of the event occurring.

The work most related to ours is $\mathrm{V_{ET}PLC}$ [12], which infers events from traces based on value changes then uses Perfume to mine temporal invariants to uncover time intervals between different events. $\mathrm{V_{ET}PLC}$ uses mined invariants to generate timed event sequences that serve as inputs for automated safety vetting PLC code. Our approach uses Perfume to find time bounds for the data invariants from the execution traces.

VI. Conclusions and Future Work

In this paper, we propose a novel approach to test PLC programs by mining invariants from execution traces. Our approach dynamically detects combined data-time invariants to observe the time performance of implementation programs. The derived invariants expressing the runtime behavior of the PLC programs are checked with the specifications. We evaluate our approach in a representative real-life scenario. The evaluation results show that our approach is useful for discovering implementation errors that are difficult to find using existing methods.

The dynamic detection technique proposed in this paper to derive data-time invariants is dedicated to the test of PLC programs. Since timing constraints are essential in many embedded systems or cyber-physical systems, we plan to apply this paper's method to test a broader class of systems beyond the scope of industrial control systems as one of the future works.

Acknowledgment. This work is partially supported by NKRDP (2019YFB2102602).

References

[1] M. Rausch and B. H. Krogh, "Formal verification of PLC programs," in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*, vol. 1. IEEE, 1998, pp. 234–238.
[2] O. Pavlovic, R. Pinger, and M. Kollmann, "Automated formal verification of PLC programs written in IL," in *Conference on Automated Deduction (CADE)*, 2007, pp. 152–163.
[3] V. Gourcuff, O. De Smet, and J.-M. Faure, "Improving large-sized PLC programs verification using abstractions," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 5101–5106, 2008.
[4] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.
[5] "NuSMV: a new symbolic model checker," http://nusmv.fbk.eu/.
[6] "UPPAAL Home," http://www.uppaal.org/.
[7] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 326–336.
[8] H. Simon and S. Kowalewski, "Mode-aware concolic testing for PLC software," in *International Conference on Integrated Formal Methods*. Springer, 2018, pp. 367–376.
[9] J. Provost, J.-M. Roussel, and J.-M. Faure, "Generation of single input change test sequences for conformance test of programmable logic controllers," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1696–1704, 2014.
[10] D. Darvas, I. Majzik, and E. B. Viñuela, "PLC program translation for verification purposes," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 61, no. 2, pp. 151–165, 2017.
[11] "The Daikon Invariant Detector User Manual," http://plse.cs.washington.edu/daikon/download/doc/daikon.html.
[12] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne *et al.*, "Towards Automated Safety Vetting of PLC Code in Real-World Plants," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 522–538.
[13] "Perfume," https://github.com/ModelInference/perfume-frontend.
[14] "MATIEC-IEC 61131-3 compiler," https://github.com/nucleron/matiec.
[15] R. Alur and T. A. Henzinger, "A really temporal logic," *Journal of the ACM (JACM)*, vol. 41, no. 1, pp. 181–203, 1994.
[16] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[17] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
[18] "Supplementary Material," https://figshare.com/s/6603090ab26eb5b0e1f6.
[19] G. Sallai, D. Darvas, and E. Blanco, "Testing, simulation, and visualisation of plc programs using x86 code generation," *CERN, Techical report EDMS*, vol. 1844850, 2017.
[20] J. Xiong, G. Zhu, Y. Huang, and J. Shi, "A User-Friendly Verification Approach for IEC 61131-3 PLC Programs," *Electronics*, vol. 9, no. 4, p. 572, 2020.