Symbolic Execution of Programmable Logic Controller Code

Shengjian Guo
Virginia Tech
Blacksburg, VA, USA

Meng Wu
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Programmable logic controllers (PLCs) are specialized computers for automating a wide range of cyber-physical systems. Since these systems are often safety-critical, software running on PLCs need to be free of programming errors. However, automated tools for testing PLC software are lacking despite the pervasive use of PLCs in industry. We propose a symbolic execution based method, named SymPLC, for automatically testing PLC software written in programming languages specified in the IEC 61131-3 standard. SymPLC takes the PLC source code as input and translates it into C before applying symbolic execution, to systematically generate test inputs that cover both paths in each periodic task and interleavings of these tasks. Toward this end, we propose a number of PLC-specific reduction techniques for identifying and eliminating redundant interleavings. We have evaluated SymPLC on a large set of benchmark programs with both single and multiple tasks. Our experiments show that SymPLC can handle these programs efficiently, and for multi-task PLC programs, our new reduction techniques outperform the state-of-the-art partial order reduction technique by more than two orders of magnitude.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation*; Software testing and debugging; Software evolution;

KEYWORDS

Symbolic execution, Test generation, Partial order reduction, Programmable logic controller, PLC, SCADA

ACM Reference format:
Shengjian Guo, Meng Wu, and Chao Wang. 2017. Symbolic Execution of Programmable Logic Controller Code. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106245

1 INTRODUCTION

Programmable logic controllers (PLCs) are specialized computers for automating electro-mechanical processes in a wide variety of industrial applications, including factory assembly lines, transportation systems, and smart power grids. PLCs are often equipped with domain-specific operating systems and virtual machines for executing software code written in programming languages such as Structured Text (ST), Ladder Diagram (LAD), and Sequential Function Chart (SFC). Since PLC software control critical infrastructures (e.g., the SCADA systems), design defects or implementation bugs may lead to catastrophes. However, despite the already widespread use of PLCs, automated testing tools are still lacking. In this work, we fill the gap by developing a *symbolic execution based* tool for automatically testing PLC software.

Symbolic execution is a popular technique for generating test inputs to systematically explore feasible paths of a program. Although symbolic execution has been applied to many programming languages, prior to this work, it has never been applied to PLCs. One reason is that PLC software are written in specialized and somewhat archaic languages that differ from mainstream programming languages, thus lacking open-source development tools. Another reason is that PLC software are periodic programs that often do not terminate, and they involve multiple tasks running concurrently with respect to each other. Tasks have different priority levels, where high-priority tasks may preempt low-priority tasks, but not vice versa. Thus, precise modeling of this *non-conventional* execution semantics is difficult.

We solve these problems by leveraging an open-source PLC compiler named *Matiec* [39] and a symbolic execution tool named *Cloud9* [19]. First, we leverage *Matiec* to translate each PLC task from the original language (e.g., ST) to C. The C code is functionally-equivalent in that each of its program paths has a corresponding path in the original PLC task, which ensures that tests generated from the C code can be mapped back to the PLC. Second, we automatically synthesize a test harness (i.e., the *main()* function in C) to invoke PLC tasks as threads. Threads are further constrained to precisely model the priority-based preemptive scheduling as defined in the PLC program semantics. Finally, we extend *Cloud9* to symbolically execute the multi-threaded C model. The new symbolic execution procedure systematically generate test cases to cover both paths of each periodic task and their interleavings.

Figure 1 shows the flow of SymPLC, where P denotes the PLC program, and translation from P to C is implemented in the *Matiec* PLC compiler. Our symbolic execution procedure based on *Cloud9* produces test cases of the form (in, sch), where in denotes the input data and sch denotes the interleaving schedule. Since *Cloud9* only supports coarse-grained thread scheduling, we extended it to execute multithreaded C code at a finer granularity. Furthermore, we propose several PLC-specific reduction techniques that leverage the periods and priorities of tasks as well as visited states to efficiently pruning redundant interleavings. Since these redundant interleavings are due to PLC-specific program semantics, they cannot be removed by partial order reduction techniques [23, 33, 49].

One advantage of SymPLC as a tool is the flexibility resulted from its separation of the *modeling* and *analysis* phases. In the modeling phase, it focuses on capturing the semantics of a PLC program written in various languages by constructing the functionally-equivalent C model. Each PLC language may be handled by a dedicated front-end; multiple front-ends may be developed independently. In the end, PLC tasks, regardless of which languages they

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106245

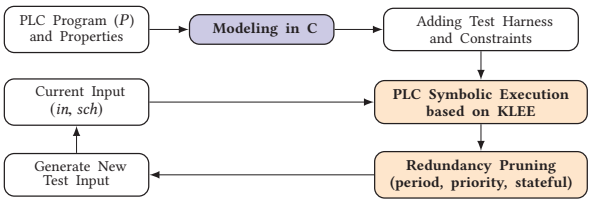

Figure 1: *SymPLC*: The overall flow of our method.

were written in, are merged to the same C model that simulates the preemptive scheduling. In the analysis phase, SymPLC focuses on executing the C model efficiently, without worrying about PLC language complications. The overall architecture allows SymPLC to easily support new languages and execution platforms.

Another advantage of SymPLC is the efficiency resulted from the PLC-specific interleaving reduction techniques. Since these new techniques are designed specifically for the PLC task scheduling, they are more effective than generic partial order reduction (POR) techniques. In the experiments section, we will show POR is often ineffective for removing redundant executions in PLC programs due to their semantic differences from thread interleavings. For example, in standard multithreaded programs, two threads with the same priority level are allowed to preempt each other, whereas in PLC programs, they are not allowed to preempt each other. Furthermore, PLC tasks are executed periodically, which means they never terminate. Our new reduction techniques are designed to take advantage of these unique characteristics.

SymPLC is a test input generation tool. As such, it differs from existing tools for simulating, verifying, or synthesizing PLC software. Specifically, simulators [13, 31, 44] can execute PLC code in controlled environments, but they require the users to handcraft test inputs. In contrast, SymPLC automatically generates these inputs. Verification tools [22, 34, 43] are designed to formally prove the correctness of properties in *models* of PLC software, but these formal models are at a much higher level of abstraction than the actual software code. In contrast, SymPLC directly executes the actual PLC code. Synthesis tools [17, 18] have the ambitious goal of generating PLC code directly from formal specifications, thus bypassing the programmers completely. However, these tools only synthesize small programs with single tasks due to scalability problems. In comparison, SymPLC is more scalable and can uniformly handle both single- and multi-task PLC programs.

We have implemented SymPLC and evaluated it on 93 PLC benchmark programs, including 49 single-task programs and 44 multi-task programs. In total, they consist of 26,713 lines of ST code, which translate to 62,926 lines of C code. Properties are expressed as assertions embedded in the source code. During our experiments, we evaluated the execution time of SymPLC as well as its effectiveness in detecting property violations. We also compared our PLC-specific reduction techniques with state-of-the-art POR techniques; for comparison, we implemented the DPOR algorithm [23] in SymPLC. Our experimental results show that SymPLC can efficiently generate test cases for all benchmark programs, and for multi-task PLC programs, in particular, our new reduction techniques significantly outperform the state-of-the-art POR technique.

To summarize, we make the following contributions:

- We develop a symbolic execution tool for PLC software by first translating the original PLC tasks to C code and then applying symbolic execution to generate the test inputs.
- We propose PLC-specific reduction techniques for more effectively eliminating redundant interleavings than state-of-the-art POR techniques.
- We implement and evaluate our techniques on a large number of benchmark programs to demonstrate their efficiency and effectiveness.

The remainder of this paper is organized as follows. First, we illustrate the main problems of testing PLC software in Section 2. Then, we present our new method for modeling the PLC program using a multi-threaded C in Section 3. We present the overall symbolic execution algorithm in Section 4, which is followed by the PLC-specific reduction techniques in Section 5. Our experimental evaluation is presented in Section 6. We review the related work in Section 7. Finally, we give our conclusions in Section 8.

2 MOTIVATING EXAMPLES

In this section, we use examples to illustrate bugs in PLC programs and explain why our new method is necessary to detect them.

2.1 Single-task PLC Programs

Figure 2 shows three PLC programs that implement a two-player game named *Responder* [16], where $I0.0$, $I0.1$ and $I0.2$ are inputs from the game host and two players, while $Q0.0$ and $Q0.1$ are outputs for the players. The program consists of two sections: `CONFIGURATION` and `PROGRAM`. The `CONFIGURATION` section declares global variables and allocates resource (CPU) to a task. For example, Task `T1` is started every 10 milliseconds and each time it executes an instance named `Game` of the program `ProgA`. The actual code of `ProgA`, provided in the `PROGRAM` section, has two statements. The first statement at Line 12 reads from $I0.0$, $I0.1$, $Q0.0$, and $Q0.1$ and then computes the new value for $Q0.0$, while the second statement computes the new value for $Q0.1$.

Initially, all inputs, outputs, and global variables are set to *false*. The host starts the game by setting $I0.0$ to *true*. Then, the players try to respond as quickly as possible by setting their inputs to *true*. If the first player is faster, its output $Q0.0$ becomes *true*, indicating she has won. But if the first player is slower, the second player's output $Q0.1$ becomes *true*. After a player's output becomes *true*, it should remain true until the host sets $I0.0$ back to *false*.

The program in Figure 2 (a) is buggy because, when both players respond at the same time, the program is not able to set both outputs to *true* (indicating a tie). Instead, it is biased toward the first player – since the PLC program is executed sequentially, i.e., one line after another, $Q0.0$ will be set to *true* first, which prevents $Q0.1$ from being set to *true* subsequently.

To fix this bug, we could introduce two auxiliary global variables $M0.0$ and $M0.1$ as shown in Figure 2 (b), to buffer the temporary outputs before assigning them to $Q0.0$ and $Q0.1$, respectively. Thus, setting $M0.0$ to *true* does not prevent $M0.1$ from becoming *true*. Indeed, when the two players respond at the same time, both outputs will be set to *true*. Unfortunately, the revised program is still faulty. Assume that both outputs have been set to *true* at the end of the first task execution because two players responded concurrently. Since task `T1` executes periodically, during the next task execution, $Q0.1$ being *true* will force $Q0.0$ to become *false*, and $Q0.0$ being *true* will force $Q0.1$ to become *false*. Thus, both outputs become *false* at

```
1  CONFIGURATION PLC_Cell1
2    VAR_GLOBAL
3      I0.0: BOOL; I0.1: BOOL; I0.2: BOOL;
4      Q0.0: BOOL; Q0.1: BOOL;
5    END_VAR
6    RESOURCE CPU_Responder ON CPU001
7      TASK T1 (INTERVAL := t#10ms, PRIORITY := 1);
8      PROGRAM Game WITH T1 : ProgA;
9    END_RESOURCE
10 END_CONFIGURATION
11 PROGRAM ProgA
12   Q0.0 := (I0.1 OR Q0.0) AND (NOT Q0.1) AND I0.0 ;
13   Q0.1 := (I0.2 OR Q0.1) AND (NOT Q0.0) AND I0.0 ;
14 END_PROGRAM
```

(a) The initial (buggy) implementation

```
1  VAR_GLOBAL
2  ...; M0.0: BOOL; M0.1: BOOL;
3  END_VAR
4  ...
5  PROGRAM ProgA
6    M0.0 := (I0.1 OR Q0.0) AND (NOT Q0.1) AND I0.0 ;
7    M0.1 := (I0.2 OR Q0.1) AND (NOT Q0.0) AND I0.0 ;
8    Q0.0 := M0.0;
9    Q0.1 := M0.1;
10 END_PROGRAM
```

(b) Revised but still buggy implementation

```
1  PROGRAM ProgA
2    M0.0 := (I0.1 AND (NOT Q0.1) OR Q0.0) AND I0.0 ;
3    M0.1 := (I0.2 AND (NOT Q0.0) OR Q0.1) AND I0.0 ;
4    Q0.0 := M0.0;
5    Q0.1 := M0.1;
6  END_PROGRAM
```

(c) The correct implementation

Figure 2: Three implementations of PLC *Responder* in ST.

the end of the second execution, which is not expected. Recall that the expected behavior is that both outputs remain true, until the host ends the game.

To fix the second bug, we need to revise the code as shown in Figure 2 (c). Compared with the program in Figure 2 (b), the modification is actually minor – we simply enlarge the scope of the two logical-OR operators to include $Q0.0$ and $Q0.1$. Because of this modification, after $Q0.0$ and $Q0.1$ become *true*, they will remain *true* during all subsequent executions of T1 regardless of the new input data, until the host ends the game by setting $I0.0$ to *false*.

These three examples show that even a simple PLC program with a single task may have subtle bugs in its implementation due to the non-conventional program semantics. Thus, automated testing tools such as SymPLC would be invaluable.

2.2 Multi-task PLC Programs

Figure 3 shows a PLC program with two tasks that implement a simplified version of the robotic controller from [15]. The RESOURCE section contains the two tasks, both of which are assigned to the device CPU001. Task T1 has a shorter period (100ms) and a higher priority, while task T2 has a longer period (200ms) and a lower priority. In PLCs, high-priority tasks may preempt low-priority tasks, but not vice versa. Assume tasks never miss their deadlines, then implicitly, the timing constraint is that T1 finishes its execution within 100ms and T2 finishes within 200ms. Furthermore, the tasks are associated with ProgA and ProgB defined below.

The PROGRAM sections provide the source code of the tasks, which share two global variables. In addition, ProgA reads from the input variable Sensor_input, whereas ProgB does not read from any primary input.

```
1  CONFIGURATION PLC_Cell2
2    VAR_GLOBAL
3      Obstacle : BOOL := 0; Forward : INT := 50;
4    END_VAR
5    RESOURCE CPU_main ON CPU001
6      TASK T1 (INTERVAL := t#100ms, PRIORITY := 1); //High
7      TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //Low
8      PROGRAM Fast WITH T1 : ProgA;
9      PROGRAM Slow WITH T2 : ProgB;
10   END_RESOURCE
11 END_CONFIGURATION
12
13 PROGRAM ProgA
14   VAR_INPUT
15     Sensor_input : INT;
16   END_VAR
17   Obstacle := 0;
18   IF (Sensor_input <= 10) THEN
19     Obstacle := 1;
20     Forward := -100;
21   END_IF;
22 END_PROGRAM
23
24 PROGRAM ProgB
25   IF (Obstacle = 0) THEN
26     Forward = 100;
27   END_IF;
28 END_PROGRAM
```

Figure 3: A Multi-task PLC Program in Structured Text.

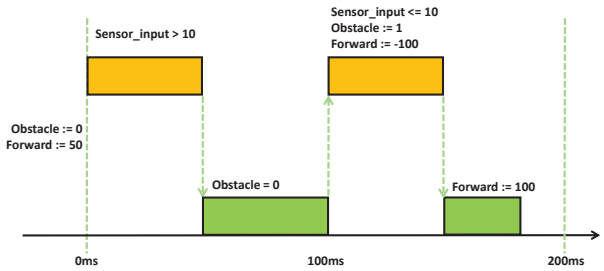

Figure 4: The task interleaving that fails the assertion.

ProgA is responsible for obstacle detection, e.g., by setting Forward to the reverse speed -100 when the value of the input Sensor_input indicates an obstacle ahead. ProgB computes the forward speed of the robot if no obstacle is detected. Thus, both tasks may write to the variable Forward (Lines 20 and 26). The race condition would cause a problem in the following scenario:

- T1 runs first and Sensor_input is greater than 10;
- T1 finishes its first execution of ProgA;
- T2 starts and proceeds to the statement at Line 26, then it is preempted by T1 before writing to Forward;
- T1 detects an obstacle and sets Forward to -100, and finishes its second execution of ProgA;
- T2 continues the execution of ProgB.

At this moment, the value of Forward is -100, and should have remained -100, but ProgB overwrites it to 100 as illustrated by Figure 4. The erroneous value is not expected, and may result in the robot hitting the obstacle.

Note that detecting the kind of bug shown in Figure 4 is not easy, since it requires a combination of the right input data (Sensor_input being > 10 in the first execution of ProgA and ≤ 10 in the second execution of ProgA) and task interleaving (ProgB is preempted by the second execution of ProgA right before the write to Forward). Although in practice, simulators may be used to reproduce this bug after it is detected, the users are required to handcraft the

error-triggering input data in the first place, which is difficult. Furthermore, simulators do not have the capability of systematically exploring the space of task interleavings. Our SymPLC tool, in contrast, solves the problem by automatically exploring the combined input and interleaving space. Thus, given the source code of this PLC program, SymPLC will generate not only the failure-triggering test data but also the corresponding task schedule.

3 MODELING PLC PROGRAM SEMANTICS

We first present our method for translating PLC tasks to equivalent C code, and then model their execution semantics using threads.

3.1 Translating PLC Tasks to C

Variables. PLC programs have different variable types. For example, the keyword VAR_INPUT defines read-only input variables, VAR_OUTPUT defines output-only variables, and VAR_EXTERNAL defines the global variables. There are eight such usage types in IEC 61131-3 standard, all of which are mapped by SymPLC to proper variables in the C program. The translation is mostly straightforward except for inputs, which require special handling.

Inputs. Variables such as `sensor_i1` and `sensor_i2` at Line 15 in Figure 5 are primary inputs. They need to be fed a symbolic value every time the corresponding task is activated. This is accomplished by calling the API function `symplc_mk_symbolic`, which returns a symbolic value for the variable. We also apply value-range constraints over these symbolic values to ensure that they always concretize to values allowed by their types. The use of symbolic values simulates the fact that input data may be arbitrary.

Timers. The behavior of PLC timers is abstracted by treating the output of each timer invocation as a symbolic variable: it is either *true* or *false* since both values are possible at run time. It ensures that actions depending on different timer outputs are always covered. Although this modeling approach may introduce potentially redundant test cases, it has the advantage of not missing any valid test input. Furthermore, we shall show that the redundant test cases may be eliminated by our new PLC-specific reduction techniques implemented inside the symbolic execution procedure.

Statements. The translation of PLC program statements from the ST language to C is straightforward because as a programming language, C is strictly more expressive than ST. Thus, any ST statement in the original program can be expressed by a corresponding C statement. Furthermore, since the number of built-in functions in ST (library functions) is fairly small, each of these functions may be replaced by a corresponding C function. In our implementation, the translation from ST code to C code is carried out by the *Matiec* PLC compiler, which has been designed to conform to the popular IEC 61131-3 standard. In Figure 5, for example, the program statements of the PLC robotic controller are translated into the C code at Lines 1-13.

3.2 Constructing the Test Harness

The test harness is the *main()* function that treats PLC tasks as threads and incorporates them to a complete C program. In Figure 5, for example, the test harness consists of Lines 14-38. There are two separate issues in simulating PLC tasks using threads. The first one is constructing a thread for potentially multiple invocations of each task (Lines 14-24). The second one is using these threads to simulate the periodic execution of PLC tasks (Lines 25-38).

```
1  bool Obstacle = 0; int Forward = 50;
2  void ProgA (int Sensor_input){
3    Obstacle = 0;
4    if (Sensor_input <= 10){
5      Obstacle = 1;
6      Forward = -100;
7    }
8  }
9  void ProgB (){
10   if (!Obstacle){
11     Forward = 100;
12   }
13 }
14 void thread1 () {
15   int sensor_i1, sensor_i2;
16   symplc_mk_symbolic(&sensor_i1, ...);
17   symplc_mk_symbolic(&sensor_i2, ...);
18   ProgA(sensor_i1);
19   //symplc_task_boundary();
20   ProgA(sensor_i2);
21 }
22 void thread2 () {
23   ProgB();
24 }
25 int main( void ){
26   pthread_t t1, t2;
27   for (i=0; i<MAX_ITER; i++) {
28     //symplc_hyperperiod_begin();
29     pthread_create(&t1, 0, thread1, 0);
30     pthread_create(&t2, 0, thread2, 0);
31     //symplc_set_priority_n_period(t1, 1, 100);
32     //symplc_set_priority_n_period(t2, 2, 200);
33     pthread_join(&t1);
34     pthread_join(&t2);
35     //symplc_hyperperiod_end();
36     assert(Obstacle == (Forward == -100)); // property
37   }
38 }
```

Figure 5: The Multithreaded C Model of the ST Program.

It is always feasible to simulate PLC task interleaving semantics using threads because threads have strictly more *permissive* interleaving semantics. That is, all possible interleavings allowed by PLC tasks are included in the set of interleavings allowed by threads. However, threads may allow certain interleavings that are not possible in PLCs. Thus, we need to constrain the threads in our C model to make the modeling of PLC tasks precise. Toward this end, the first step is to construct all threads for a hyper-period.

Hyper-period. PLC tasks in the same program may have different periods. For instance, in our running example, `T1` has a period of 100ms and `T2` has a period of 200ms. In this context, a hyper-period is defined as the least common multiplier of the periods of all tasks. Thus, the hyper-period of our running example is 200ms. Clearly, within a hyper-period, `T1` will be executed twice and `T2` will be executed once. The reason why we are interested in the hyper-period is because timing-related program behaviors repeat themselves after each hyper-period. Thus, focusing on analyzing the tasks within each hyper-period is important. Furthermore, the hyper-period will be used to reduce the symbolic execution cost. In the C model, we construct one thread for all the execution instances of each task in a hyper-period. That is why in Figure 5, `thread1()` invokes `ProgA` twice, but `thread2()` invokes `ProgB` only once.

Periodic execution. Next, we construct a for-loop in the *main()* function to execute all threads concurrently. Each iteration of the for-loop corresponds to a hyper-period. The total number of iterations is bounded by a user-defined parameter MAX_ITER, since PLC programs in general are non-terminating programs. Within each hyper-period, we first create the threads and then set their parameters (period and priority). These parameters will be passed to the symbolic execution engine to avoid exploring interleavings

that are not allowed by the PLC program semantics. As shown in Figure 5, we use special API functions to signal the boundary of the hyper-period and boundaries of tasks within each thread.

Assertions. The assertion at the end of the hyper-period represents the property to be checked. In PLC programs, developers may use the ASSERTION(. . .) keyword to specify a property. Such assertions are translated into assertions in the C program straightforwardly. SymPLC also allows its user to specify additional assertions, which are inserted at the end of the hyper-period (e.g., at Line 36 in Figure 5). Assertions are reachability properties because each *assert(c)* may be modeled as *if(!c) ERROR*, where *ERROR* is an error location. During symbolic execution, if any error location is reached, the symbolic execution tool produces an error-triggering test case.

4 SYMBOLIC EXECUTION

In this section, we formally define PLC programs and then present the overall symbolic execution algorithm.

4.1 Multithreaded C Model

The multithreaded C model of a PLC program consists of a set of periodic tasks $\mathsf{T} = \{T_1, \ldots, T_n\}$. Each task $T_i \in \mathsf{T}$, where $1 \leq i \leq n$, denotes an instance of a PLC program within a hyper-period. Consider the program named `ProgA` in Figure 5, which has two instances in a hyper-period (Lines 18 and 20). In our C model, these two instances are considered as different tasks in T.

Tasks share a set GV of global variables. Each T_i also has a set LV_i of local variables. In addition, each T_i may read from a set PI of primary inputs. Thus, T_i can be viewed as a sequential program that reads from primary inputs as well as global variables, updates the global variables, and computes the outputs. Since tasks are executed periodically, in addition to being a sequential program, each T_i has the following attributes:

- $T_i.tid$ denotes the unique identifier of the task;
- $T_i.priority$ denotes the priority level of the task;
- $T_i.period$ denotes the execution period of the task within a hyper-period;
- $T_i.startT$ denotes the start time of the task's period;
- $T_i.endT$ denotes the end time of the task's period.

Due to PLC's non-conventional interleaving semantics, for any two tasks T_i and T_j, where $i \neq j$,

- if $T_i.priority < T_j.priority$, then T_j may preempt the execution of T_i at any time between $T_i.startT$ and $T_i.endT$, but T_i cannot preempt T_j;
- if $T_i.priority = T_j.priority$, neither task may preempt the other task.

This is different from the standard interleaving semantics of a multithreaded program, where threads with the same priority are allowed to preempt each other.

The execution of task T_i leads to a sequence of events $t_1, \ldots, t_k$. For ease of presentation, we assume each event $t \in T_i$ inherits all attributes of the task T_i including *tid*, *priority*, *period*, *startT*, and *endT*. In other words, $t.startT$ and $t.endT$ are the expected start time and end time of the period of the task T_i. In addition, we introduce $t.task$ to denote the task T_i that generates the event t.

Some events in a PLC program are reads and writes of global variables, while others are computations over local variables. Local operations are further divided into branching statements e.g., $if(c)$, and assignments $lv = exp$, where exp may be arithmetic computations, bit-string operations, boolean operations, etc. Without loss of generality, we assume $if(c)$ involves only local variables, because $if(exp(gv))$, where $gv \in \mathsf{GV}$, can always be replaced by $lv = gv$; $if(exp(lv))$, where $lv \in \mathsf{LV}_i$ is a newly added local variable and $if(exp(lv))$ involves only local variables. Thus, during symbolic execution, we only need to consider two types of events:

- *interleaving schedule* events, which perform context switches right before global reads and writes;
- *sequential computation* events, which are either $if(c)$ or assignments over local variables.

Only *interleaving schedule* events may affect the execution order. Thus, we will focus on analyzing them to identify redundant interleavings. In contrast, *sequential computation* events are handled in the same way as in standard symbolic execution tools.

Algorithm 1 Symbolic execution of a multi-task PLC program.

```
    Initially: State stack S = {};
    Run SymPLC(s_0) where s_0 is the initial state.
 1: SymPLC( State s ) {
 2:     S.push( s );
 3:     if (s is an interleaving schedule node)
 4:         foreach ( event t that is enabled and ¬Redundant(s, t) )
 5:             s' ← NextState (s, t);
 6:             SymPLC(s');
 7:     else if (s is a sequential computation node)
 8:         foreach ( event t whose path condition is satisfiable )
 9:             s' ← NextState (s, t);
10:             SymPLC(s');
11:     else
12:         output test case if s is the end of an execution
13:     S.pop();
14: }
15: NextState( State s, Event t ) {
16:     s.sel ← t;
17:     Compute new symbolic state s' based on s and t;
18:     return s';
19: }
20: Redundant( State s, Event t ) {
21:     if (t is redundant according to the theory of POR)
22:         return true;
23:     return false;
24: }
```

4.2 Overall Algorithm

Algorithm 1 shows the overall procedure, which closely follows prior techniques for symbolic execution of multithreaded programs [6, 19, 25, 26]. Here, S is a stack of symbolic states. Each symbolic state $s \in S$ is a tuple $\langle pcon, \mathcal{M}, enabled, sel \rangle$, where $pcon$ is the path condition, $\mathcal{M}$ is the symbolic memory, $enabled$ is the set of enabled events, and sel is the event executed at s.

Initially, SymPLC starts with the symbolic state s_0. Then, depending on the type of the current state s, it either schedules a context switch or executes a sequential computation. Specifically, if s is an interleaving schedule node (right before a global read or write), SymPLC is invoked recursively to explore each possible schedule together with the subsequent events (Lines 4-6). If s is a sequential computation node (local statement within a task), SymPLC is invoked recursively to explore each branch and assignment (Lines 8-10). Upon reaching the end of an execution (Lines 11-12), SymPLC generates the corresponding test case and backtracks from the current state.

Subroutine NextState takes the current state s and the event t as input, and returns the newly computed symbolic state s' as output. For brevity, we omit the details of this symbolic execution

Algorithm 2 Deciding if event t chosen at s is redundant.

```
1:  Redundant ( State s, Event t) {
2:      if (t is redundant according to the DPOR algorithm)
3:          return true;
4:      // Priority-based reduction
5:      let t' be the last event in S before reaching t;
6:      if (t' is NULL)
7:          if (t.priority is not the highest in s.enabled) return true;
8:      else
9:          if (t is about to preempt t')
10:             if (t'.priority ≥ t.priority) return true;
11:             if (t'.startT ≥ t.startT) return true;
12:         // Period-based reduction
13:         if(t'.tid ≠ t.tid)
14:             if (t'.endT < t.startT) return true;
15:             if (t'.startT ≥ t.endT) return true;
16:         if (t is the last event in t.task)
17:             if (∃t_h, t_l ∈ S that t_h.tid == t.tid and t_h preempted t_l)
18:                 if (∃t'' ∈ S that t''.task interleaved with t.task)
19:                     if (t''.startT ≥ t_l.endT) return true;
20:     return false;
21: }
```

process since it remains the same as in standard symbolic execution procedures in the literature.

The challenge is mitigating the combinatorial blowup associated with the event interleavings (Lines 4-6) because, in the worst case, the number of interleavings is exponential in the number of global operations. Traditional techniques for mitigating the interleaving explosion are based on *partial order reduction (POR)* [23, 30, 33, 46, 49], which is to group interleavings into equivalence classes and then pick a representative interleaving from each equivalence class while skipping the other (redundant) interleavings. In Algorithm 1, this is implemented inside Subroutine Redundant. However, POR does not consider the additional interleaving constraints imposed by PLC tasks. As such, it is not effective in mitigating the interleaving explosion problem in PLC programs.

5 OUR PLC-SPECIFIC REDUCTIONS

In this section, we present three new reduction techniques designed to take advantage of the unique characteristics of PLC programs. Specifically, they are related to leveraging information from (1) the priorities of tasks, (2) periods of tasks, and (3) previously visited program states during symbolic execution.

Algorithm 2 shows our implementation of the first two reductions. The third reduction will be presented in Section 5.3. Here, the subroutine Redundant returns *true* if executing t from the state s is redundant, whether it is due to DPOR or infeasibility according to the PLC interleaving semantics. Within the current hyper-period, we define t' to be the last event chosen before reaching s (Line 5). In the subsequent two sections, we illustrate how these two types of reductions make use of t' in more details.

5.1 Priority-based Reduction

In this new reduction, we impose three rules which directly follow the way PLCs schedule their tasks:

(1) The active task with the highest priority must be scheduled to before other active tasks whenever a hyper-period starts.
(2) A running task can only be preempted by another running task with a strictly higher priority;
(3) If a high-priority task starts before the period beginning of a low-priority task, there must be no interleavings between these two tasks.

```
1  RESOURCE CPU_main ON CPU001
2    TASK T1 (INTERVAL := t#200ms, PRIORITY := 1); //High
3    TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //Low
4    PROGRAM Fast WITH T1 : ProgA;
5    PROGRAM Slow WITH T2 : ProgB;
6  END_RESOURCE
```

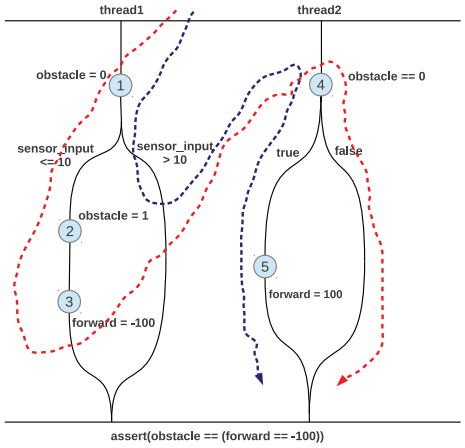

Figure 6: The control flow graph of the modified program.

We encode these rules into Lines 5-11 of Algorithm 2. First, when both high-priority task and low-priority task are enabled and ready to run, the PLC should always run the high-priority task first. This corresponds to the conditions at Lines 5-6: if t' does not exist, it means t is the first event in the current hyper-period. At this moment, the PLC must choose the highest-priority task to execute. Thus, if t is not the highest-priority task, Redundant returns *true*.

On the other hand, if t' exists and t is about to preempt t', we first leverage the task priorities to perform a reduction, and then leverage both the priorities and the periods to perform another reduction. Specifically, we check two the following conditions.

The first condition at Line 10 ensures that t has a strictly higher priority than t', because PLCs only allow high-priority tasks to preempt low-priority tasks but not vice versa. And tasks with the same priority are not allowed to preempt each other. The second condition at Line 11 makes use of periods of the tasks. Note that at this point, we know t's priority is higher than that of t'. The condition checks if the (expected) start time of the period of t is before the (expected) start time of the period of t'. If this is the case, the interleaving is infeasible because the low-priority event t' should not have occurred before t (it should only be executed after the end of t's period).

Consider the PLC program in Figure 3 again as an example, but with an important modification—setting the `INTERVAL` of `T1` to `t#200ms` instead of `t#100ms`. Since both tasks now need `t#200ms`, the hyper-period becomes `200ms`, meaning `ProgA` and `ProgB` are invoked once each in the new threads `thread1` and `thread2`, respectively. The control flow of these two new threads are shown in Figure 6, where nodes are the global reads or writes and solid lines are the control flows. Recall that the primary input `Sensor_input` is modeled as a *symbolic* variable, thus allowing both branches immediately after the node 1 to be taken. In contrast, the branches immediately after the node 4 depend only on the value of the global variable `Obstacle`.

Table 1: Interleavings explored by priority-based reduction.

ID	All-Interleavings	DPOR	SymPLC	ID	All-Interleavings	DPOR	SymPLC
1	1-4-5	yes	yes	7	4-5-1	yes	
2	1-2-3-4	yes	yes	8	4-1-5		
3	1-2-4-3			9	4-1-2-5-3	yes	
4 *	1-4-2-3-5	yes		10 *	4-1-2-3-5	yes	
5	1-4-2-5-3	yes		11	4-1-5-2-3		
6	1-4-5-2-3	yes		12	4-5-1-2-3	yes	

If it were a standard multithreaded program, each thread would be allowed to preempt the other one at the control flow nodes, thus leading to a total of 12 interleavings, as shown in the second and fifth columns of Table 1, labeled *All-Interleavings*. Among them, the two interleavings marked with * would violate the assertion. After applying the DPOR algorithm, for example, eight interleavings would remain while the other four would be removed. Specifically, 1-2-4-3 is removed because it is equivalent to 1-2-3-4; 1-4-5-2-3 is equivalent to 1-4-2-5-3; 4-1-5 is equivalent to 4-5-1; and 4-1-5-2-3 is equivalent to 4-5-1-2-3.

However, applying our new priority-based reduction would lead to significantly fewer interleavings. In fact, only two interleavings would remain, which are shown by the red and blue dotted lines in Figure 6. This is because, according to our second rule, all six interleavings in Column 2 except 1-4-5 and 1-2-3-4 are infeasible, because the low-priority task (T2) preempts the high-priority task. Similarly, according to our first rule, all six interleavings in Column 6 are infeasible, because when both T1 and T2 are active and ready to run at the beginning, the PLCs would always choose to execute the high-priority task (T1).

Since the erroneous interleavings (4 and 10) are not explored by SymPLC, and SymPLC terminates after two hyper-periods (due to the termination condition to be presented in Section 5.3), we have proved the validity of this assertion condition.

Our implementation uses an *on-the-fly* computation to decide whether the current interleaving is feasible. Take the second rule as an example. Whenever an instruction accessing global variables is interpreted in the symbolic execution engine, we check the priority of its task against the operation history of current execution. If a preceding operation is from an active task whose priority is higher than the current one, then the interleaving resulted from executing t at s should be skipped. The first and the third rule are developed in a similar fashion.

In Figure 6, for instance, 4-5-1 is determined to be infeasible immediately after the first node 4 is reached by SymPLC, since the first rule is violated. Therefore, SymPLC backtracks from node 4 while skipping the interleavings numbered 8-12 entirely.

5.2 Period-based Reduction

In this new reduction, we develop two rules over task interleaving:

(1) Two tasks are allowed to interleave only when their expected execution periods overlap in time;
(2) If a high-priority task T_h preempts a low-priority task T_l, T_h must not interleave with any task whose period begin time is not earlier than the period end time of T_l.

We implement these rules at Lines 13-19 of Algorithm 2. Recall that $t.startT$ and $t.endT$ are the expected logical time when the period of t begins and ends (we are not concerned with the actual start time and end time of t, except that they must fall within the period). Without these rules, any two operations from different

```
1  CONFIGURATION PLC_Cell1
2    RESOURCE CPU_main ON CPU001
3      TASK T1 (INTERVAL := t#100ms, PRIORITY := 1); //H-priority
4      TASK T2 (INTERVAL := t#200ms, PRIORITY := 2); //M-priority
5      TASK T2 (INTERVAL := t#300ms, PRIORITY := 3); //L-priority
6      PROGRAM Fast WITH T1 : ProgA;
7      PROGRAM Const WITH T2 : ProgB;
8      PROGRAM Slow WITH T2 : ProgC;
9    END_RESOURCE
10 END_CONFIGURATION
```

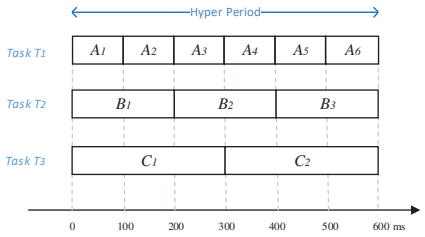

Figure 7: Three periodic tasks with a hyper-period of 600ms.

threads would have been allowed to execute concurrently in the same hyper-period. However, since each task must meet its own deadline, some of them can never run concurrently.

Consider the program in Figure 7 as our example, which has three tasks T1, T2 and T3 with periods 100ms, 200ms and 300ms, respectively. Thus, the hyper-period is 600ms, allowing T1 to execute six times, T2 to execute three times, and T3 to execute twice. For ease of presentation, let the six instances of T1 be denoted from A_1 to A_6, the three instances of T2 be denoted from B_1 to B_3, and the two instances of T3 be denoted C_1 and C_2.

Without the timing-related information, symbolic execution would have to explore all possible interleavings of these tasks, including the obviously infeasible ones between A_1 and B_2, for example, which do not overlap in time. These infeasible interleavings will be removed by applying our reduction rules.

We first compare the task IDs of t' and t in Algorithm 2 – different IDs means they belong to different tasks. The next rule at Line 14 is straightforward, since interleaving cannot occur if the two tasks do not overlap in time. In our running example, the period of A_1 is [0ms, 100ms] while the period of B_2 is [200ms, 400ms]. Obviously, events in A_1 do not occur concurrently with events in B_2. Similarly, the periods of B_3 and C_1 do not overlap. Both of these two cases are handled by the conditions at Lines 14-15 of Algorithm 2.

The second rule (Lines 16-19) is more subtle because the infeasible interleavings are deduced via a preceding interleaving, based on both periods and priorities of involved tasks. As shown in Figure 7, the period B_2 is expected to start before A_4. Thus, it appears that A_4 may interleave with B_2. However, if B_2 preempts C_1 in a particular execution, then B_2 must end before the end of the period of C_1, to allow C_1 to meet its deadline. Since B_2 would have ended before the start of the period of A_4, it cannot run concurrently with A_4. Thus, in this particular example, A_4 and B_2 can no longer interleave.

This example also illustrates the third reduction rule in Section 5.1: A_3 starts from the 200ms, while the earliest time B_2 can start is 200ms. Since T1 has a higher priority, and A_3 starts earlier than B_2, the execution of A_3 cannot be interrupted by B_2. Thus, any interleaving between them is guaranteed to be infeasible.

Algorithm 3 Next state computation with stateful reduction.

```
1: NextState( State s, Event t ) {
2:     s.sel ← t;
3:     Compute the new symbolic state s′ based on s and t;
4:     if (t is plc_hyperperiod_end)
5:         if (s′ ⊆ visited) return NULL;
6:         else visited ← visited ∪ {s′};
7:     return s′;
8: }
```

5.3 Stateful Exploration

Now we present the state-based reduction. Recall PLC tasks are periodic and thus never terminate. Furthermore, symbolic execution by default is geared toward detecting bugs as opposed to proving the correctness of properties. Thus, applying SymPLC with a user-specified depth bound in general will never prove the absence of bugs in a PLC program. However, information of already-visited states may be leveraged to detect early-termination conditions. This allows SymPLC to drastically reduce the number of test cases, as well as prove the correctness of properties.

Algorithm 3 shows the modified NextState subroutine in Algorithm 1 that implements this method. At the end of each hyper-period, it checks if the new symbolic state s' has been visited previously. If the answer is yes, it returns NULL instead of s' which forces SymPLC to backtrack immediately.

In general, the state of a PLC program is a valuation of all variables as well as program counters (PC) of all tasks. However, since we are concerned with the program state only at the end of a hyper-period (where all tasks have ended and local variables are out of the scope), only the valuation of global variables needs to be considered.

Let R be the set of all reachable states of a PLC program at the end of the hyper-period. Ideally, SymPLC should generate enough test cases to cover all states in R. We will show through experiments that, due to the nature of these PLC programs, the termination condition can often be met after a few hyper-periods. It also means SymPLC should be designed to terminate as soon as the symbolic execution procedure stops generating previously unexplored states.

Consider a program named *IndustrialAuto*4 from [18], which contains a state machine whose state variable, `CSTATE6`, may take a number of values. A brute-force application of SymPLC would result in exponentially many program paths as the number of hyper-periods increases. For example, after five hyper-periods, the number of executions becomes 3176. In contrast, applying our new stateful reduction decreases the total number of executions down to 45. Furthermore, since the symbolic execution procedure detects the early-termination condition after 3 hyper-periods, all unfalsified properties are considered to be formally proved.

6 EXPERIMENTS

We have implemented SymPLC based on the *Matiec* PLC compiler [39] and the *Cloud9* symbolic virtual machine [19]. We used *Matiec* to translate ST code of each PLC task to ANSI C, and then created a test harness to incorporate these tasks. We implemented the test harness generator using Python. The resulting multithreaded C model was then executed by the extended *Cloud9*, which uses KLEE [12] internally for symbolic execution. We extended *Cloud9* to handle the PLC-specific program features.

Our experiments answer the following research questions: (1) Can SymPLC efficiently handle both single-task and multi-task PLC programs? Is SymPLC effective in detecting property violations as well as proving their correctness? (2) Are the PLC-specific reduction techniques (stateful, period, and priority) effective in reducing the search space? Do they outperform state-of-the-art POR techniques? For comparison purposes, we implemented the state-of-the-art dynamic partial-order reduction (DPOR) algorithm [23, 33, 49] in SymPLC to identify and remove redundant interleavings.

We evaluated SymPLC on two sets of benchmark programs. The first set consists of 49 single-task PLC programs collected from various online sources [16, 18, 29]. The second set consists of 44 multi-task PLC programs that implement several embedded controllers [14, 15]. Each PLC program has 30 to 3,418 lines of ST code, which translate to 90 to 8,783 lines of C code. In total, they consist of 26,713 lines of ST code, which translate to 62,926 lines of C code. The C code is first compiled to LLVM bitcode and then symbolically executed by the modified *Cloud9*. Correctness properties are expressed as assertions embedded in the programs. We conducted all our experiments on a computer with a 3.40 GHz CPU and 8 GB RAM running Ubuntu 12.04 Linux.

6.1 Results on Single-task PLC Applications

Table 2 shows the experimental results on single-task PLC programs. Since each hyper-period has one task, the number of iterations is the same as the number of tasks executed. In this table, Columns 1–3 show the statistics of each benchmark program, including the name, the number of lines of original ST code, and the number of lines of generated C code. Columns 4-8 show the detailed results of SymPLC, including the maximum number of iterations reached (#.Iter), whether stateful reduction detected convergence (Conv), the number of tests generated, execution time in seconds, and the instruction coverage (#.ICov). The last three columns show the assertion checking results, including the number of undecided, falsified, and proved assertions.

If SymPLC finds an execution that fails an assertion, the assertion is falsified. If SymPLC does not find such an execution before reaching early termination, the assertion is proved. Otherwise, the assertion remains undecided.

Although symbolic execution is geared toward falsifying assertions, Table 2 shows that our stateful reduction is also effective in detecting termination conditions. As a result, SymPLC can prove 154 assertions (in addition to falsifying 34 assertions) and there are only 18 undecided assertions. In contrast, without stateful reduction, there would be 172 undecided assertions.

Furthermore, the number of iterations ranges from 2 to 14, indicating that repeatedly executing the same PLC tasks after that many hyper-periods does not lead to new program states. Instead, the main difficulty resides in covering the input space, which is what symbolic execution is designed for.

The average *Instruction Coverage* for all benchmarks is 89.7%, which did not reach 100% even for benchmarks that converged, apparently because some of these instructions are unreachable.

6.2 Results on Multi-task PLC Applications

In this section, we show the performance differences between non-stateful and stateful exploration inside SymPLC, and then compare the various interleaving reduction techniques.

Table 3 shows the results on multi-task PLC programs. Columns 1-3 show the benchmark name and statistics of the hyper-period, including the total number of tasks and global operations executed

Table 2: Results of SymPLC on single-task PLC programs.

Program	LOC		#.Iter	Conv	#.Tests	Time (s)	#.ICov (%)	Assertions		
	ST	C						Undet.	Falsified	Proved
G4LTL_ST1	470	1,249	5	Y	305	36.2	89.1	0	0	1
G4LTL_ST2	188	504	5	Y	316	13.2	87.5	0	0	1
G4LTL_ST3	111	252	6	Y	116	4.9	87.3	0	0	2
G4LTL_ST4	1,409	4,279	7	Y	1,498	140.2	44.5	0	1	1
G4LTL_ST5	321	855	5	Y	240	9.6	97.8	0	0	2
G4LTL_ST6	69	154	2	Y	67	2.5	98.5	0	2	1
G4LTL_ST7	86	156	2	Y	272	1231.9	78.2	0	0	1
G4LTL_ST8	488	1,661	8	Y	368	16.3	74.4	0	0	5
G4LTL_ST9	577	914	10	Y	686	69.7	92.6	0	1	1
G4LTL_ST10	257	435	6	Y	354	27.8	99.8	0	1	2
IndustrialAuto1	45	145	2	Y	12	0.6	95.1	0	0	2
IndustrialAuto2	43	150	2	Y	10	0.5	95.5	0	2	2
IndustrialAuto3	206	379	4	Y	289	10.6	99.7	0	1	3
IndustrialAuto4	65	172	3	Y	45	1.7	98.9	0	0	3
IndustrialAuto5	105	273	3	Y	65	2.1	98.8	0	1	7
IndustrialAuto6	126	276	6	Y	25	1.0	84.1	0	0	9
IndustrialAuto7	126	275	5	Y	34	1.3	92.4	0	0	8
IndustrialAuto8	199	485	13	Y	55	3.1	60.0	0	0	11
IndustrialAuto9	2,444	8,291	14	Y	143	8.8	11.7	0	2	9
IndustrialAuto10	1,195	3,266	12	No	5,084	>3600	34.3	15	1	0
IndustrialAuto11	75	218	3	Y	23	0.9	97.8	0	0	8
IndustrialAuto12	1,580	3,781	6	Y	3,216	813.1	99.9	0	0	14
IndustrialAuto13	255	607	8	Y	130	4.9	61.5	0	1	6
IndustrialAuto15	3,418	8,783	8	Y	1,349	151.8	65.7	0	1	11
IEC-1	30	90	2	Y	6	0.4	83.2	0	0	1
IEC-2	53	135	2	Y	619	25.6	97.9	0	1	1
IEC-3	118	260	2	No	8,041	>3600	98.4	2	2	0
IEC-4	66	173	6	Y	216	9.1	92.5	0	0	3
IEC-5	32	72	2	Y	6	0.3	88.2	0	0	2
IEC-6	52	140	2	Y	306	9.9	98.0	0	0	1
IEC-7	173	474	3	Y	37	2.5	85.1	0	1	1
LD-Program1	89	136	3	Y	1,084	64.5	79.1	0	0	4
LD-Program2	336	403	2	No	10,508	>3600	67.0	1	0	0
LD-Program3	92	135	11	Y	231	10.3	99.6	0	1	1
LD-Program4	110	150	2	Y	601	29.9	74.9	0	1	2
Mixer	181	251	4	No	5,088	>3600	88.6	0	2	0
Evaporator	178	238	2	Y	287	11.9	81.7	0	1	3
Hydraulic	118	128	2	Y	54	2.4	83.0	0	0	4
Safe	215	313	2	Y	1,724	199.5	92.0	0	0	2
Logic	234	322	2	Y	125	6.2	78.9	0	1	8
Lift	187	169	2	Y	160	8.1	71.6	0	0	2
Plastic	187	215	2	Y	360	22.6	75.1	0	0	2
Bargraph	126	143	4	Y	4,316	429.6	98.0	0	0	2
Jedyka	80	92	7	Y	132	4.5	99.5	0	2	0
Glowny	70	86	4	Y	62	2.1	93.5	0	2	1
IL-Tool	137	171	2	Y	362	20.4	99.2	0	1	2
Shutter	83	125	4	Y	643	49.6	94.3	0	3	0
Alarm	68	107	4	Y	326	22.4	99.5	0	1	1
Fountain	50	95	9	Y	339	28.7	99.6	0	1	1
Total	16,923	42,183			50,335	17,913		**18**	**34**	**154**

Table 3: Results of SymPLC on multi-task PLC programs.

Program	Hyper-period		Non-Stateful						Stateful					
	#.Task	#.Ops	#.Iter	#.Test	#.Time	Assertions			#.Iter	#.Test	#.Time	Assertions		
						Und	Fal	Pro				Und	Fal	Pro
nxt2.prog1	3	16	10	10	0.2	1	0	0	3	3	0.4	0	0	1
nxt2.prog2	3	16	10	1027	2.8	0	1	0	2	7	0.4	0	1	0
nxt2.prog3	5	27	10	59048	372.2	1	0	0	2	5	0.4	0	0	1
nxt2.prog4	7	37	10	59767	>600	1	0	0	2	5	0.4	0	0	1
nxt2.prog5	5	28	4	109,669	>600	0	1	0	2	43	0.7	0	1	0
nxt2.prog6	7	38	4	71,631	>600	0	1	0	2	43	0.8	0	1	0
nxt2.prog7	7	38	3	66,907	>600	1	0	0	2	91	1.4	0	0	1
nxt3.prog1	5	15	5	43,313	>600	1	0	0	3	24	0.6	0	0	1
nxt3.prog2	7	19	4	57,396	>600	0	1	0	2	93	1.2	0	1	0
nxt3.prog3	8	28	4	15,080	>600	1	0	0	2	20	0.8	0	0	1
nxtway01	6	42	5	41,629	>600	1	0	0	2	25	0.6	0	0	1
nxtway02	6	46	3	35,449	>600	0	1	0	3	149	2.4	0	1	0
nxtway03	9	62	3	33,809	>600	0	1	0	3	978	16.2	0	1	0
nxtway04	9	66	4	26,988	>600	0	1	0	2	575	10.4	0	1	0
nxtway05	6	38	9	11122	>600	1	0	0	3	11	0.6	0	0	1
nxtway06	6	34	4	46,580	>600	0	1	0	3	860	12.1	0	1	0
nxt.pi00	6	46	5	21,808	>600	0	1	0	2	55	1.2	0	1	0
nxt.pi01	6	46	4	23,348	>600	1	0	0	2	98	1.7	0	0	1
nxt.pi02	8	62	3	21,139	>600	0	1	0	2	368	6.5	0	1	0
nxt.pi03	5	38	4	25,406	>600	0	1	0	3	179	2.9	0	1	0
trans01	6	41	3	26,367	>600	0	1	0	3	502	7.9	0	1	0
trans02	6	41	6	498	>600	1	0	0	4	27	0.9	0	0	1
trans03	6	39	4	33,572	>600	0	1	0	5	2,638	38.7	0	1	0
trans04	6	41	4	29,326	>600	1	0	0	3	73	1.4	0	0	1
trans05	9	59	5	11582	>600	1	0	0	3	19	0.8	0	0	1
attend01	6	35	5	34,658	>600	1	0	0	4	64	1.2	0	0	1
attend02	6	42	3	32,932	>600	0	1	0	3	388	5.9	0	1	0
attend03	6	48	4	20,723	>600	1	0	0	2	119	2.3	0	0	1
attend04	6	39	4	23,537	>600	1	0	0	3	101	1.9	0	0	1
att4_01	7	40	3	23,655	>600	1	0	0	3	855	14.5	0	0	1
att4_02	7	33	4	32,505	>600	1	0	0	3	105	1.9	0	0	1
race01	6	47	3	21,990	>600	0	1	0	2	166	2.6	0	1	0
race02	6	38	5	17,730	>600	0	1	0	3	41	0.9	0	1	0
race03	9	63	3	14,960	>600	1	0	0	2	275	6.7	0	0	1
nobadmode01	6	34	4	22,915	>600	1	0	0	3	88	1.5	0	0	1
nobadmode02	7	45	4	10,840	>600	1	0	0	3	86	2.3	0	0	1
nobadmode03	6	52	3	14,051	>600	1	0	0	3	614	13.1	0	0	1
nobadmode04	6	51	5	12,666	>600	1	0	0	3	102	2.2	0	0	1
ctm01	7	123	7	47067	>600	1	0	0	9	131	2.2	0	0	1
ctm02	7	120	5	82,330	>600	1	0	0	4	178	2.5	0	0	1
ctm03	6	115	9	72,135	>600	1	0	0	5	82	1.2	0	0	1
aso_01	11	67	2	20,269	>600	1	0	0	2	438	12.4	0	0	1
aso_02	9	49	2	19,204	>600	1	0	0	2	90	2.9	0	0	1
aso_03	9	53	3	26,610	>600	0	1	0	2	452	9.4	0	1	0
Total				1,423,248	24,975	**27**	**17**	**0**		11,266	199	**0**	**17**	**27**

in each hyper-period, because they are closely related to the complexity of the interleaving exploration. Columns 4-9 show results of SymPLC without stateful reduction, including the maximum number of iterations reached, the number of test cases generated, the run time, and the assertion checking results. Columns 10-15 show results of SymPLC with stateful reduction. We set the time bound to 10 minutes and hyper-period iteration bound to 10.

Since non-stateful SymPLC cannot detect convergence, it does not prove properties. In contrast, stateful SymPLC can prove properties. Our results show that stateful SymPLC only needed a few hyper-periods to detect convergence. In contrast, non-stateful SymPLC frequently timed out or generated more test cases (1.4 million versus 11K). Both detected 17 violations, but stateful SymPLC also proved 27 assertions, whereas non-stateful SymPLC did not.

Table 4 shows the result of comparing different interleaving reduction techniques. Here, KLEE denotes the default symbolic execution algorithm in *Cloud9* augmented with the capability of handling threads. DPOR denotes the enhanced version of KLEE where we added the implementation of dynamic partial order reduction. Among the three PLC-specific reductions, *Period* denotes our period-based reduction technique, *Priority* denotes our priority-based reduction technique, and *Period+Priority* denotes the full-blown implementation of our reduction in SymPLC. All methods shown in Table 4 were used in conjunction with the stateful reduction. For each individual method, we show the number of test cases generated and the total execution time in seconds. Since the time limit was set to 10 minutes, >600s means the corresponding method was forced to terminate after running out of time.

As shown in the total numbers in the last row, the full-blown reduction implemented in SymPLC, denoted *(Period+Priority)*, significantly outperformed KLEE and DPOR, two state-of-the-art symbolic execution techniques. Specifically, the reduction in the number of test cases is more than two orders of magnitude. Furthermore, the full-blown reduction is significantly more efficient than *Period*-based reduction (11,266 versus 1,433,944) or *Priority*-based reduction (11,266 versus 267,352) alone. This means applying both *Period* and *Priority* based reductions has led to synergistic impact.

Table 4: Results of comparing the reduction techniques.

Program	KLEE [12]		DPOR [23]		PLC-specific Reductions					
					Period		Priority		Period+Priority	
	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)	#. Test	Time (s)
nxt2.prog1	43,960	>600	135	1.6	135	1.6	3	0.4	3	0.4
nxt2.prog2	44,200	>600	19	0.5	19	0.5	7	0.4	7	0.4
nxt2.prog3	44,595	>600	5,644	62.2	3,839	39.5	24	0.6	5	0.4
nxt2.prog4	45,683	>600	47,652	>600	35,531	452.3	40	0.7	5	0.4
nxt2.prog5	50,067	>600	52,666	>600	47,422	>600	144	1.5	43	0.7
nxt2.prog6	44,442	>600	46,521	>600	45,867	>600	528	5.1	43	0.8
nxt2.prog7	44,383	>600	46,280	>600	45,394	>600	827	9.8	91	1.4
nxt3.prog1	47,159	>600	48,111	>600	44,560	>600	250	3.3	24	0.6
nxt3.prog2	52,490	>600	54,248	>600	41,011	>600	5,792	59.9	93	1.2
nxt3.prog3	45,842	>600	54,851	>600	25,527	>600	3,117	37.8	20	0.8
nxtway01	53,425	>600	43,799	>600	41,003	>600	195	2.7	25	0.6
nxtway02	51,317	>600	45,304	>600	45,633	>600	1,646	22.2	149	2.4
nxtway03	51,609	>600	35,932	>600	39,562	>600	35,445	>600	978	16.2
nxtway04	60,785	>600	45,557	>600	38,916	>600	31,045	>600	575	10.4
nxtway05	51,589	>600	38,140	>600	39,562	>600	713	10.1	11	0.6
nxtway06	46,392	>600	50,025	>600	45,756	>600	4,608	68.6	860	12.1
nxt_pi00	49,470	>600	39,158	>600	40,716	>600	561	8.2	55	1.2
nxt_pi01	49,978	>600	40,446	>600	39,212	>600	1,226	16.9	98	1.7
nxt_pi02	48,670	>600	28,824	>600	29,922	>600	10,216	169.9	368	6.5
nxt_pi03	43,048	>600	40,384	>600	39,884	>600	239	3.7	179	2.9
trans01	40,869	>600	42,269	>600	40,684	>600	5,170	78.9	502	7.9
trans02	40,893	>600	38,720	>600	39,364	>600	99	2.1	27	0.9
trans03	36,714	>600	43,546	>600	43,602	>600	11,112	168.6	2,638	38.7
trans04	38,568	>600	35,123	>600	25,535	>600	357	5.3	73	1.4
trans05	49,880	>600	46,138	>600	35,480	>600	4,207	69.3	19	0.8
attend01	43,298	>600	54,320	>600	57,849	>600	222	3.1	64	1.2
attend02	32,541	>600	56,541	>600	18,866	>600	20,292	12.8	388	5.9
attend03	46,143	>600	35,023	>600	35,585	>600	343	5.1	119	2.3
attend04	45,951	>600	36,224	>600	32,570	>600	432	6.5	101	1.9
att4_01	47,312	>600	40,691	>600	36,861	>600	2,391	37.2	855	14.5
att4_02	46,969	>600	34,278	>600	40,327	>600	364	5.5	105	1.9
race01	45,610	>600	38,334	>600	37,424	>600	500	7.2	166	2.6
race02	45,277	>600	17,756	231.9	2,236	32.2	176	3.9	41	0.9
race03	44,974	>600	22,145	>600	32,658	>600	28,112	>600	275	6.7
nobadmode01	49,094	>600	46,622	>600	40,932	>600	314	4.6	88	1.5
nobadmode02	43,561	>600	51,877	>600	6,616	144.6	16,862	346.7	86	2.3
nobadmode03	43,525	>600	49,719	>600	48,408	>600	2,745	49.9	614	13.1
nobadmode04	43,962	>600	42,283	>600	37,739	>600	2,558	46.5	102	2.2
ctm01	51,345	>600	51,810	>600	776	11.4	846	10.9	131	2.2
ctm02	55,711	>600	63,599	>600	54,294	>600	621	7.8	178	2.5
ctm03	49,343	>600	51,875	>600	41,839	>600	97	1.3	82	1.2
aso_01	42,260	>600	39,386	>600	10,582	>600	20,607	>600	438	12.4
aso_02	44,403	>600	43,847	>600	12,461	>600	29,191	>600	90	2.9
aso_03	44,235	>600	40,646	>600	11,785	>600	23,108	>600	452	9.4
Total	**2,041,542**	26,400	**1,786,468**	24,296	**1,433,944**	22,822	**267,352**	4,895	**11,266**	199

7 RELATED WORK

Since PLCs are widely used in industry control applications, there exist some integrated development environments (IDEs) and simulators for PLCs. However, they are designed primarily for mimicking the behavior of PLC devices on host computers. Although simulators may be used to test a PLC program, the user must handcraft the test inputs. As we mentioned earlier, manually creating high-quality test inputs is difficult. Furthermore, even with the test inputs, it is still necessary to explore the possible task schedules under these inputs. Unfortunately, simulators are not equipped to perform this task. SymPLC fills the gap by leveraging symbolic execution to automatically generate high-quality test inputs, as well as systematically cover the possible interleavings.

There is also a large body of work on formal verification of PLC applications [2, 5, 7–9, 40, 42, 43]. In this context, a formal model of the target PLC has to be constructed before it is analyzed by verification tools such as UPPAL [41] and NuSMV [1, 5]. Various optimizations are also proposed to increase the efficiency of these verification tools [27, 43, 47, 48]. However, there are several fundamental differences between these model checkers and SymPLC.

First, constructing and tuning formal models are not easy. They require expertise in formal methods and the application domains, thus limiting the practical use. Second, formal models are at higher abstraction levels than the actual code; thus, they are more suitable for checking design defects [20, 36, 43] than implementation bugs. Finally, none of the existing tools handles multi-task PLC programs; indeed, they focus exclusively on single-task programs, perhaps to avoid the difficulty in modeling the concurrency semantics.

In contrast, SymPLC requires no formal model; instead, it relies on symbolic execution to directly checking the PLC software code. SymPLC also uniformly models both single-task and multi-task PLC programs. While symbolic execution has been routinely used for testing sequential and concurrent programs written in a wide variety of programming languages, to the best of our knowledge, it has never been applied to multi-task PLCs before. Bohlender et al. [11] applied concolic testing to single-task PLC programs but did not consider the interleaving of multiple tasks.

Our modeling of preemptive scheduling semantics is related to testing and verifying periodic programs. In this particular context, Regehr et al. [45] used threads to simulate the behavior of interrupt-driven C programs. Kroening et al. [32] verified C code with nested interrupts using a bounded model checker. Chaki et al. [14, 15] also developed several tools for verifying periodic C programs. Although there are similarities, these works are significantly different because the semantics of PLC tasks differs from both interrupts and threads. Furthermore, none of these prior works was related to symbolic execution, which is the focus of our work.

Our method for restricting task interactions based on priorities was inspired by techniques for model checking real-time software [4, 10, 21, 28, 37], which encode necessary conditions of the scheduler semantics to increase fidelity and reduce state-space explosion. Similar approaches were also used in combination with symbolic execution [38]. Our stateful reduction can be viewed as an instance of the state-merging and matching technique in symbolic execution [3, 25, 26, 35]. However, none of the existing techniques has been applied to PLC software.

There are techniques for synthesizing PLC software from specifications. For example, Cheng et al. [17, 18] synthesized PLC code from linear temporal logic specifications. Gelen et al. [24] synthesized PLC code for real-time supervisory control of a manufacturing system. However, due to inherent limitations, so far, they can only produce small programs. SymPLC can be considered as a complementary testing method to these program synthesis tools.

8 CONCLUSIONS

We have presented a symbolic execution tool for automatically testing single- and multi-task PLC programs. It takes the PLC source code as input, translates it into C code, and then applies symbolic execution. As such, it can systematically explore feasible paths of individual PLC tasks as well as their interleavings. Toward this end, our main contribution is developing a number of PLC-specific reduction techniques for eliminating redundant interleavings. Our experiments show that the tool is efficient in handling a large number of PLC benchmark programs. On multi-task PLC programs, in particular, our new reduction techniques significantly outperform the state-of-the-art partial order reductions technique.

ACKNOWLEDGMENTS

This material is based upon research supported in part by the U.S. National Science Foundation under grants CNS-1617203 and CNS-1702824 and the U.S. Office of Naval Research under award number N00014-13-1-0527.

REFERENCES

[1] Borja Fernandez Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Trans. Industrial Informatics*, 11(6):1400–1410, 2015.

[2] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 184–200, 1998.

[3] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *International SPIN Workshop on Model Checking Software*, pages 163–181, 2006.

[4] Luciano Baresi, Carlo Ghezzi, and Luca Mottola. On accurate automatic verification of publish-subscribe architectures. In *International Conference on Software Engineering*, pages 199–208, 2007.

[5] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. Regression verification for programmable logic controller software. In *International Conference on Formal Methods and Software Engineering*, pages 234–251, 2015.

[6] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 491–506, 2014.

[7] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Counterexample-guided abstraction refinement for PLCs. In *International Workshop on Systems Software Verification*, 2010.

[8] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 338–341, 2012.

[9] Sebastian Biallas, Mirco Giacobbe, and Stefan Kowalewski. Predicate abstraction for programmable logic controllers. In *Formal Methods for Industrial Critical Systems - 18th International Workshop*, pages 123–138, 2013.

[10] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim Guldstrand Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 106–114, 2008.

[11] Dimitri Bohlender, Hendrik Simon, Nico Friedrich, Stefan Kowalewski, and Stefan Hauck-Stattelmann. Concolic test generation for PLC programs using coverage metrics. In *International Workshop on Discrete Event Systems*, pages 432–437, 2016.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

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

[14] Sagar Chaki, Arie Gurfinkel, and Nishant Sinha. Efficient verification of periodic programs using sequential consistency and snapshots. In *International Conference on Formal Methods in Computer-Aided Design*, pages 51–58, 2014.

[15] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-bounded analysis of real-time systems. In *International Conference on Formal Methods in Computer-Aided Design*, pages 72–80, 2011.

[16] Gang chen, Xiaoyu Song, and Ming Gu. PLC program verification and analysis using the coq theorem prover. *Acta Scientiarum Naturalium Universitatis Pekinensis*, 46(1):30–34, 2010.

[17] Chih-Hong Cheng, Yassine Hamza, and Harald Ruess. Structural synthesis for GXW specifications. In *International Conference on Computer Aided Verification*, pages 95–117, 2016.

[18] Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess, and Stefan Stattelmann. G4LTL-ST: automatic generation of PLC programs. In *International Conference on Computer Aided Verification*, pages 541–549, 2014.

[19] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[20] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In *Integrated Formal Methods - 12th International Conference*, pages 508–522, 2016.

[21] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, and Gurdip Singh. Model-checking middleware-based event-driven real-time embedded software. In *International Symposium on Formal Methods for Components and Objects*, pages 154–181, 2002.

[22] Jean-Marie Farines, Max Hering de Queiroz, Vinicius G. da Rocha, Ana Maria M. Carpes, François Vernadat, and Xavier Crégut. A model-driven engineering approach to formal verification of PLC programs. In *IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8, 2011.

[23] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[24] Gökhan Gelen and Murat Uzam. The synthesis and PLC implementation of hybrid modular supervisors for real time control of an experimental manufacturing system. *Journal of Manufacturing Systems*, 33(4):535–550, 2014.

[25] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 531–542, 2016.

[26] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[27] C. G. Haba, R. Cociu, and L. Cociu. Mixed mode verification of PLC based control systems. In *International Symposium on Advanced Topics in Electrical Engineering*, pages 1–4, 2011.

[28] Matthew Hoosier, Matthew B. Dwyer, Robby, and John Hatcliff. A case study in domain-customized model checking for real-time component software. In *International Symposium on Leveraging Applications of Formal Methods*, pages 161–180, 2004.

[29] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids.* Springer Science & Business Media, 2010.

[30] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[31] Lock-Jo Koo, Chang Mok Park, Chang Ho Lee, SangChul Park, and Gi-Nam Wang. Simulation framework for the verification of PLC programs in automobile industries. *International Journal of Production Research*, 49(16):4925–4943, 2011.

[32] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference*, pages 229–234, 2015.

[33] Markus Kusano and Chao Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[34] E. V. Kuzmin, Valery A. Sokolov, and D. A. Ryabukhin. Construction and verification of PLC-programs by LTL-specification. *Automatic Control and Computer Sciences*, 49(7):453–465, 2015.

[35] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–204, 2012.

[36] Tim Lange, Martin R. Neuhäußer, and Thomas Noll. Speeding up the safety verification of programmable logic controller code. In *International Haifa Verification Conference*, pages 44–60, 2013.

[37] Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In *International Symposium on Automated Technology for Verification and Analysis*, pages 444–456, 2005.

[38] Kasper Søe Luckow, Corina S. Pasareanu, and Bent Thomsen. Symbolic execution and timed automata model checking for timing analysis of Java real-time systems. *EURASIP J. Emb. Sys.*, 2015:2, 2015.

[39] IEC 61131-3 compiler. URL: https://www.openhub.net/p/matiec.

[40] Stephen E. McLaughlin, Saman A. Zonouz, Devin J. Pohly, and Patrick D. McDaniel. A trusted safety verifier for process controller code. In *Network and Distributed System Security Symposium*, 2014.

[41] Houda Bel Mokadem, Béatrice Bérard, V. Gourcuff, O. De Smet, and J. Roussel. Verification of a timed multitask system with uppaal. *IEEE Trans. Automation Science and Engineering*, 7(4):921–932, 2010.

[42] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In *Formalisms for Reuse and Systems Integration*, pages 55–78. 2015.

[43] Johanna Nellen, Kai Driessen, Martin Neuhäußer, Erika Ábrahám, and Benedikt Wolters. Two CEGAR-based approaches for the safety verification of PLC-controlled plants. *Information Systems Frontiers*, pages 1–26, 2016.

[44] Sang C. Park, Chang Mok Park, Gi-Nam Wang, Jonggeun Kwak, and Sungjoo Yeo. PLCStudio: Simulation based PLC code verification. In *Proceedings of the 2008 Winter Simulation Conference*, pages 222–228, 2008.

[45] John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electr. Notes Theor. Comput. Sci.*, 174(9):139–150, 2007.

[46] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[47] Mo Xia, Mian Sun, Guiming Luo, and Xibin Zhao. Design and implementation of automatic verification for PLC systems. In *IEEE International Conference on Cognitive Informatics and Cognitive Computing*, pages 374–379, 2013.

[48] Litian Xiao, Mengyuan Li, Ming Gu, and Jiaguang Sun. A hierarchy framework on compositional verification for PLC software. In *IEEE International Conference on Software Engineering and Service Science*, pages 204–207, 2014.

[49] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.