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:
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...
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 intricacies. 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 \( I_0.0, I_0.1 \) and \( Q_0.0, Q_0.1 \) are inputs from the game host and two players, while \( Q_0.0 \) and \( Q_0.1 \) 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 \( I_0.0, I_0.1, Q_0.0, \) and \( Q_0.1 \) and then computes the new value for \( Q_0.0 \), while the second statement computes the new value for \( Q_0.1 \).

Initially, all inputs, outputs, and global variables are set to false. The host starts the game by setting \( I_0.0 \) 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 \( Q_0.0 \) becomes true, indicating she has won. But if the first player is slower, the second player’s output \( Q_0.1 \) becomes true. After a player’s output becomes true, it should remain true until the host sets \( I_0.0 \) 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, \( Q_0.0 \) will be set to true first, which prevents \( Q_0.1 \) from being set to true subsequently.

To fix this bug, we could introduce two auxiliary global variables \( M_0.0 \) and \( M_0.1 \) as shown in Figure 2 (b), to buffer the temporary outputs before assigning them to \( Q_0.0 \) and \( Q_0.1 \), respectively. Thus, setting \( M_0.0 \) to true does not prevent \( M_0.1 \) 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, \( Q_0.1 \) being true will force \( Q_0.0 \) to become false, and \( Q_0.0 \) being true will force \( Q_0.1 \) to become false. Thus, both outputs become false at
Figure 2: Three implementations of PLC Responder in ST.

(a) The initial (buggy) implementation

(b) Revised but still buggy implementation

(c) The correct implementation

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).

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

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 \( T = \{ T_1, \ldots, T_n \} \). Each task \( T_i \in T \), 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, \( T_i \) 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 \( T_j.endT \), but \( T_i \) cannot preempt \( T_j \);
- if \( T_j.priority = T_i.priority \), 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.,

\[
\begin{align*}
if(c) \text{ and assignments } lv &= \exp, \text{ where } \exp \text{ may be arithmetic computations, bit-string operations, boolean operations, etc.} \\
\text{Without loss of generality, we assume } if(c) \text{ involves only local variables, because if(}exp(v))\text{, where } v \in GV, \text{ can always be replaced by } lv = gv; \text{ if(}exp(lv))\text{, where } lv \in LV \text{ is a newly added local variable and if(}exp(lv)) \text{ involves only local variables. Thus, during symbolic execution, we only need to consider two types of events:}
\end{align*}
\]

- **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.

#### 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 \( \langle pcon, M, enabled, sel \rangle \), where \( pcon \) is the path condition, \( M \) 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...
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.

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 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 t#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.

![Figure 6: The control flow graph of the modified program.](image-url)
Symbolic Execution of Programmable Logic Controller Code

Table 1: Interleavings explored by priority-based reduction.

<table>
<thead>
<tr>
<th>ID</th>
<th>All-Interleavings</th>
<th>DPOR</th>
<th>SymPLC</th>
<th>ID</th>
<th>All-Interleavings</th>
<th>DPOR</th>
<th>SymPLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-4-5</td>
<td>yes</td>
<td>yes</td>
<td>7</td>
<td>4-5-1</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>1-2-3-4</td>
<td>yes</td>
<td>yes</td>
<td>8</td>
<td>4-1-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1-2-4-3</td>
<td>yes</td>
<td>yes</td>
<td>9</td>
<td>4-1-2-5</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>1-4-2-3-5</td>
<td>yes</td>
<td>yes</td>
<td>10</td>
<td>4-1-2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1-4-2-5-3</td>
<td>yes</td>
<td>yes</td>
<td>11</td>
<td>4-1-5-2-3</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>1-4-5-2-3</td>
<td>yes</td>
<td>yes</td>
<td>12</td>
<td>4-5-1-2-3</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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 begins 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 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 A1 to A6, the three instances of T2 be denoted from B1 to B3, and the two instances of T3 be denoted C1 and C2.

Without the timing-related information, symbolic execution would have to explore all possible interleavings of these tasks, including the obviously infeasible ones between A1 and B2, 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 mean 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 A1 is [0ms, 100ms] while the period of B2 is [200ms, 400ms]. Obviously, events in A1 do not occur concurrently with events in B2. Similarly, the periods of B1 and C1 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 B2 is expected to start before A4. Thus, it appears that A1 may interleave with B2. However, if B2 preempts C1 in a particular execution, then B2 must end before the end of the period of C1, to allow C1 to meet its deadline. Since B2 would have ended before the start of the period of A4, it cannot run concurrently with A4. Thus, in this particular example, A4 and B2 can no longer interleave.

This example also illustrates the third reduction rule in Section 5.1: A3 starts from the 200ms, while the earliest time B2 can start is 200ms. Since T1 has a higher priority, and A3 starts earlier than B2, the execution of A3 cannot be interrupted by B2. Thus, any interleaving between them is guaranteed to be infeasible.
5.3 Stateful Exploration

Now we present the state-based reduction. Recall PLC tasks are periodic and thus never terminate. Moreover, 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 `IndustrialAuto4` 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, periodic, 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...
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. The timing limit was set to 10 minutes and hyper-period’s 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 PL-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.
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 hosts. 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 thread models 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 C programs [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 technique 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 reduction techniques.

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.
Symbolic Execution of Programmable Logic Controller Code

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

REFERENCES


