A Verification Approach for Programmable Logic Controllers

Xiang-yu LUO¹,a*, Yong LI¹, Wan-xiao DU¹, Fan YANG² and Zhi-gang YANG¹

¹College of Computer Science & Technology, Huaqiao University, Xiamen 361021, China
²College of Mechanical Engineering and Automation, Huaqiao University, Xiamen 361021, China

¹luoxy@hqu.edu.cn
*Corresponding author

Keywords: Programmable logic controller, System modeling, Software verification.

Abstract. This paper presents an iterative approach to the verification of programmable logic controllers. We explore the modeling method for timing, environment and controller logics in a system, in which predicate abstraction and counterexample-guided refinement strategies are employed. We use a representative example to illustrate the proposed approach and verify it by the model checker CBMC. The experimental results show the validity of the approach.

Introduction

Programmable Logic Controller systems (PLCs) are widely used in industry [1]. Typical applications range from household appliances chemical process control to railway automation systems and emergency shutdown systems in nuclear power plants. As PLCs are increasingly utilized in safety-critical systems, the correctness of PLC systems is critical. Traditional verification techniques use simulation to validate the design. Unfortunately, generating test vectors is labor. The use of formal methods is an alternative approach to validate system designs.

Application of formal methods for PLC software has been proposed [2-3]. In the previous work, formal models are extracted from the PLC programs for verification. Model checking works on a model of system with a logical specification of a desired behavior of the system model. It checks whether the model adheres to the specification by effectively searching the entire state space of the model. With the increased scale and complexity of systems, state space explosion problem imposes a major vexing problem in automated program verification [4-6]. Predicate abstraction constitutes a promising strategy of reducing state space [4]. Based on predicate abstraction, the counter-example-guided abstraction refinement (CEGAR) framework [5-7] plays a key role in application of model checking. It is successfully applied to a few software verification projects to verify device drivers, C programs and Java software [5-7].

This paper presents an iterative approach to verifying programmable logic controllers. System modeling for timing constraints, environment and controller logics is explored. Predicate abstraction and counterexample-guided refinement strategies are employed. A representative example is used to illustrate our modeling and verification process. The proposed strategy is validated by the analysis and the experiment.

The rest of the paper is structured as follows. We first introduce the architecture of PLC systems, explore the modeling method for PLC systems, then present the verification method and a case study to illustrate the verification process. Finally we conclude the paper.

The PLC Architecture

Typical PLC systems are composed of PLC controllers, I/O interface and controlled devices. A PLC controller is a digital microcomputer. It is the kernel of PLCs which implements the user’s program. It includes a global system clock which is used as the standard timer or the time-interrupting subprogram. The I/O interface includes input/output circuit, sensors, or Analog/Digit translators, etc. It collects input signals and converts them between analog and digital components. The controlled devices are implemented in physical environments. Its outputs are usually temperature, pressure,
velocity, or height. Thus, the PLC system is a typical cyber-physical system which consists of sensors and actuators to interact with microprocessors.

Consider a water tank isobaric PLC system implemented by Siemens S-200 [7]. In this example, a water tank is used to maintain a constant water pressure. Water is continuously being taken from the water tank at a varying rate \( v_1 \). A variable speed pump is used to add water to the tank at a rate \( v_0 \) that maintains adequate water pressure and also keep the tank from being emptied. The set point for this system is a water level setting that is equivalent to the tank being 80% full. The process variable Water_height is supplied by a float gauge that provides an equivalent reading of how full the tank is and that can vary from 0% (or empty) to 100% (or completely full). The output is a value of pump speed that allows the pump to run from 0% to 100% of maximum speed.

**Modeling PLC Systems**

**Modeling Clock**

We define a clock function block \( M_c \) to model the clock’s behavior shown in Fig. 1. The similar idea was proposed in [10]. The clock function block \( M_c \) simulates the execution time of program instructions and is called by timer, interrupts subprogram, or the model of controlled device. In the model, the integer \( a \) can be a tick of PLC’s CPU, the average instruction time of a special type PLC, or an actual instruction execution time. In the example, we assume that \( a = 5ms \). A function block is similar to a function, but it preserves its history in local variables that are not initialized at each function block call [8]. The statement at line 1 implements the first call of the function.

```plaintext
FUNCTION_BLOCK CLOCK
VAR
1. Clock: INT := 0;
END_VAR
2. Clock := Clock + a;
END_FUNCTION_BLOCK
```

**Figure 1. The clocking model.**

A PLC program can set and check timers. We use the timer function block to mimic the timer behavior. Fig. 2 shows an interrupt timer function block, where the line 2’s statement implements the first call of the function.

```plaintext
FUNCTION_BLOCK INTERRUPT_TIMER
VAR_INPUT
1. IT := 100;
END_VAR
VAR
2. Initial := true;
3. T0, T := Clock;
END_VAR
4. T := CLOCK();
5. IF Initial
6. T0 := T; Initial := false;
7. ELSEIF T - T0 = IT
8. Interrupt(); Initial := true;
END_FUNCTION_BLOCK
```

**Figure 2. The Interrupt-timer function block.**

**Modeling Controlled Devices**

The behavior of controlled devices is modeled as function block \( M_d \), shown in Fig. 3. It models the physical external environments. The rational number variables \( \text{in} \) represents the program output which controls the devices. The variable \( \text{out} \) represents the properties of the controlled device varied with the environment, such as temperature, pressure, velocity, or height, and it is also the program’s partial input. The function \( f \) is the physical characteristic of the controlled device.

In the example, the controlled devices are a tank and an electromotor. For simplicity, we presume that the tank height is of 32 meters, the initial value of \( v_1 \) is 32m/s when time \( t \leq 32ms \) and is 0, otherwise. We use a set of functions to simulate the behavior of the tank and electromotor. Here \( h_{\text{new}} \) is the real water level sampled by the sensor, and \( h_{\text{old}} \) is the water level of the last cycle. Without loss of generality, we simplify the electromotor and use AQW to represent water supply in the last 100ms. \( h_
is the variant of height in the last 100ms, and 0 \leq h_{\text{old}}, h_{\text{new}}, h \leq 32m. We have the following restrictions on the variables mentioned above: \quad h_{\text{new}} = h + h_{\text{old}} \quad , \quad h = AQW - v \cdot t \quad , \quad v_1 = 32m / s(t \leq 300ms) \quad , \quad v_1 = 0(t > 300ms) \quad \text{and} \quad h_{\text{old}} = 0.8 \times 32m. The function block model is shown in Fig. 4 where the line 3 implements the first call of the function.

<table>
<thead>
<tr>
<th>FUNCTION_BLOCK DEVICE</th>
<th>FUNCTION_BLOCK DEVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_INPUT</td>
<td>VAR_INPUT</td>
</tr>
<tr>
<td>In : Type;</td>
<td>1. AQW : R; END_VAR</td>
</tr>
<tr>
<td>END_VAR</td>
<td>VAR_OUTPUT</td>
</tr>
<tr>
<td></td>
<td>2. h_new : Type; END_VAR</td>
</tr>
<tr>
<td>VAR_OUTPUT</td>
<td>VAR</td>
</tr>
<tr>
<td>Out : Type;</td>
<td>3. Initial: h_{\text{old}} = 0.8 \times 32; END_VAR</td>
</tr>
<tr>
<td>END_VAR</td>
<td>4. T = CLOCK()</td>
</tr>
<tr>
<td></td>
<td>5. IF T &lt;= 300</td>
</tr>
<tr>
<td></td>
<td>6. v1 := 32;</td>
</tr>
<tr>
<td></td>
<td>7. ELSE v1 = 0;</td>
</tr>
<tr>
<td></td>
<td>8. h' := AQW - v1 \cdot 100;</td>
</tr>
<tr>
<td></td>
<td>9. h_{\text{new}} := h' + h_{\text{old}};</td>
</tr>
<tr>
<td></td>
<td>10. h_{\text{old}} := h_{\text{new}};</td>
</tr>
<tr>
<td></td>
<td>END_FUNCTION_BLOCK</td>
</tr>
<tr>
<td>Out := f (In, Clock);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. The controlled device's model. Figure 4. The water tank's function block.

The Holistic Model

We build the holistic model \( M \) in C-like language shown in Fig. 5. Instructions between line 1 and 2 initials function block clock, interrupt-timer and controlled device. Line 2 simulates the cyclic behavior of PLCs. Lines 3, 12 and 18 simulate the clock of PLCs. Lines 4-6 simulate the subprogram SBR_0, which initiates the PID parameters. The function timer() in lines 6 and 13 simulates the behavior of interrupt-timer. The function PidInput() in lines 10 and 17 simulates the controlled device. The function PidOutput() simulates the execution of "PID VB100, 0" in the original program.

```
1: int Clock = 0;
   bool Initial = true;
   bool SMB2 = false;
   Init(RC, h, h_{\text{old}}, h_{\text{new}}, T2DI16, T2DI18, AQW = 0;
   Init(RC, T2DI1, T2DI1A, T2DI18, SMB2 = 100;
2: While (true) {
   3:   Clock = Clock + 5;
   4:   if (SMB1 == 0) {
       5:     T = Clock;
       6:     if (Initial) {
           7:       T2DI1 = T;
           8:       Initial = false;
           9:     } else if (T2DI1 = SMB2) {
               10:     PidInput();
               11:     PidOutput();
               12:     Initial = true;
           } else {
               13:     Timer();
               14:     if (Initial) {
                   15:         T2DI1 = T;
                   16:         Initial = false;
               } else if (T2DI1 = SMB2) {
                   17:     PidInput();
                   18:     PidOutput();
                   19:     Initial = true;
               } else {
                   20:         PidInput();
                   21:         PidOutput();
                   22:         Initial = true;
               } else {
                   23:         PidInput();
                   24:         PidOutput();
                   25:         Initial = true;
               }
   7:   Clock = Clock + 5;
```

Figure 5. The Program of the example.
The Verification Process

Given a C program P and a set E of predicates, SLAM [7] automatically creates an abstraction of P as a Boolean program BP(P; E). A Boolean program is a C program with only Boolean type. The Boolean program has the same control-flow structure as P but contains only |E| Boolean variables, each representing a predicate in E. For each statement of P, SLAM automatically constructs the corresponding Boolean transfer functions that conservatively represent the effect of statement on the predicates in E. The CEGAR method is an iterative process of three stages [6, 7]: abstraction, checking and refinement.

Step 1 (Abstraction) By a finite set of global predicates, the algorithm explores the state space of the program step by step. At the same time, it creates an abstract Boolean program to find a counter-example trace. If no abstract counter-example traces are found, the safety property holds; otherwise it goes to Step 2.

Step 2 (Checking) Abstract traces are checked. When an abstract trace is found, it is checked if the counter-example corresponds to a concrete counter-example in the original program. If so, then a program error has been found; otherwise the trace is a spurious counter-example and it goes to Step 3.

Step 3 (Refinement) New local predicates are found by analyzing the spurious counter-example to eliminate the spurious counter-example. If this is the case, the exploration is resumed and the search for the program space continues.

The property \( P = \text{“if Water\_height is always bigger than 50\% of Water\_height”} \) is a safety property. It can be represented as a temporal logic CTL formula: \( AG(h\_new > 50\% * 32000) \) [14]. Safety property checking can be converted to reachability analysis [15, 16]. We insert a label “error” in M at the line 23 in Fig. 5. The safety verification checks if an “error” is reachable. To do this, we need search all the traces of M. If there is a trace leading to the “error” label, then M does not satisfy the property P.

Based on the CEGAR frame, we consider a C program as control flow automata CFA [6]. It is an abstract unfolded syntax tree representing a precise global control graph of the system. The CFA of the example is shown in Fig. 6. Each automaton node is marked by the value of the program counter and the edges are labeled by the corresponding instructions in Fig. 5. The CEGAR method is an iterative process on CFA [6].

![Figure 6. The CFA (control flow automata) of the example.](image)
Experiments and Conclusions

We use a model checker CBMC [16] to verify the property in the previous section. The experimental results are shown in Table 1. CBMC is a bounded-model checker, so we set the “while” iteration number as 1000 in line 2 of Fig. 5. We run some examples on a PC of Intel Duo CPU 2.0GHz with 2G RAM. For the Water Tank example, the verification time is 45.86 seconds. CBMC checks all 2002 traces with 19744 assignments. The quizMachine.c is a 3-quiz application and the classification.c is industry carry control case [17]. The traffic.c is a railway traffic control system extracted from [18]. It is a distributed PLC system. The elevator.c is a 6-floor, 2-elevator control system extracted from [19]. The assignment column is the enumerated program statements. The claim column is the number of checked safety assertions [19].

Table 1. The experimental results.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Assignments</th>
<th>Claims</th>
<th>Time(seconds)</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water Tank</td>
<td>19744</td>
<td>25</td>
<td>45.86</td>
<td>yes</td>
</tr>
<tr>
<td>quizMachine.c</td>
<td>102</td>
<td>8</td>
<td>1.34</td>
<td>yes</td>
</tr>
<tr>
<td>classification.c</td>
<td>394</td>
<td>34</td>
<td>2.12</td>
<td>no</td>
</tr>
<tr>
<td>traffic.c</td>
<td>785</td>
<td>65</td>
<td>5.67</td>
<td>yes</td>
</tr>
<tr>
<td>elevator.c</td>
<td>2235</td>
<td>37</td>
<td>12.94</td>
<td>yes</td>
</tr>
</tbody>
</table>

In summary, we present an iterative approach to the verification of programmable logic controllers. Predicate abstraction and counterexample-guided refinement strategies are employed for the modeling of timing, environment and controller logics in a system. The approach is validated by the experiments based on the model checker CBMC.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (Grant No. 61170028), the Program for New Century Excellent Talents in Fujian Province University (Grant No. 2013FJ-NCET-ZR03), the Promotion Program for Young and Middle-aged Teacher in Science and Technology Research of Huaqiao University (Grant No. ZQN-YX109), the Natural Science Foundation of Fujian Province of China (Grant No. 2015J01255), the Scientific Research Foundation of Huaqiao University (Grant No. 11BS108) and the Guangxi Key Laboratory of Trusted Software (Grant No. kx201323).

References


