Formalizing Ladder Logic Programs and Timing Charts for Fault Impact Analysis and Verification of Fault Tolerance

Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-23-01
January 2023

Formalizing Ladder Logic Programs and Timing Charts for Fault Impact Analysis and Verification of Fault Tolerance

Ali Ebnenasir

January 2023

Abstract

This paper presents a novel approach for modeling, automated analysis and verification of fault tolerance in Ladder Logic (LL) programs for Programmable Logic Controllers (PLCs). The goal is to provide a framework for control engineers where they can verify LL programs in the absence and in the presence of faults in a simple-to-learn and effective formal language. To enable formal modeling of faults and fault tolerance requirements, we first devise a method for transforming LL programs and timing charts to formal specifications respectively in Promela and Linear Temporal Logic (LTL). Using such a formalization, engineers can generate Promela code from LL programs and can translate the requirements of timing charts to LTL expressions. We characterize two types of requirements for LL programs, namely intracycle and intercycle requirements. As a result, engineers can simulate LL programs in the SPIN model checker and verify them for intra-intercycle requirements. We also present a method for modeling faults and analyzing the impact of faults inside each scan cycle and across several scan cycles. We then characterize the novel notions of intracycle and intercycle fault tolerance for LL programs. We demonstrate the proposed method in the context of an industrial Carriage Line system.

1 Introduction

Programmable Logic Controller (PLC) programs play a crucial role in monitoring and controlling the critical infrastructure of our society, and it is of paramount importance to develop systematic methods for the verification and synthesis of highly dependable PLC programs. The IEC 61131-3 standard [3] defines several languages for PLC programming, but the most common languages include the Ladder Logic (LL) [8] and Structured Text (ST). PLC programs have a periodic nature where in each period, a.k.a. *scan cycle*, they read from input signals, execute program logic, and actuate output signals. The duration of the scan cycle is set based on program length and logic, and it is often in the scale of tens of milliseconds (much smaller than the fastest reaction time of the physical plant under control). In addition to their periodic nature, PLC programs have timers and counters that span over several scan cycles, which makes it difficult to correctly formalize their behavior and design fault tolerance functionalities. Moreover, mainstream engineers often use timing charts to specify the requirements of PLC programs in terms of their input/output signals. While there are many approaches for formal specification and verification of PLC programs, little work has been performed on formalizing timing charts, modeling faults and designing fault-tolerant PLC programs. This paper presents an approach for (i) formalizing LL programs in a small and simple subset of Promela [1, 15]; (ii) characterizing timing chart requirements for cyclic PLC programs; (iii) specifying timing chart requirements in Linear Temporal Logic (LTL), and (iv) modeling faults and fault tolerance concerns for LL programs.

There are numerous methods that generate formal specification from PLC programs (either LL or ST programs), most of which (i) use formal languages (e.g., PetriNet) that are hard to learn for mainstream engineers; (ii) provide little to enable the formal specification of timing charts (which capture the requirements of PLC programs), and (iii) fail to model faults and fault tolerance concerns. For example, there are numerous methods [14, 21, 25] for generating PetriNet specifications from PLC programs. Brinksma and Mader [7] evaluate the verification and optimization of real-time control schedules for small size PLC programs in the SPIN and UPPAAL model checkers. Ljungkrantz *et al.* [19] present a method for formalization of LL programs in terms of reusable components augmented with pre-postconditions (i.e., contracts), and verified by the SMV model checker [23]. Kuzmin *et al.* [18] put forward a method for model checking of PLC programs in SMV. Darvas *et al.* [10] present a model checking-based method for conformance checking of PLC programs with respect to their specifications and alternative implementations of the same specifications. Darvas *et al.* [9, 20] present PLCVerif along with a set of natural language requirement patterns for model checking of ST programs. Mao *et al.* [22] propose a refinement-based approach for formalizing the requirements of PLC programs in Event-B, verifying their safety and generating ST code. Mesli-Kesraoui *et al.* [24] generate timed automata from LL programs and then verify them for CTL properties in the UPPAAL model checker. Belo Lourenço *et al.* [6] translate LL programs and timing charts into WhyML code and use the Why3 environment to verify LL programs with respect to their timing charts using theorem proving. Garcia *et al.* [13] present a formalization framework for the translation of hybrid programs (specified in differential dynamic logic) to ST code and vice versa. While existing methods present a variety of formal semantics for PLC programs, they suffer from several problems, namely (i) translating PLC programs to formal languages that are difficult to learn for mainstream engineers; (ii) lacking a systematic method for formalization of intra-cycle and inter-cycle properties of timing charts, and (iii) focusing on verification of PLC programs in the absence of faults and fault tolerance aspects.

We present a formal semantics for LL programs in a small and simple subset of the Promela [1, 15] modeling language, and devise a method for formalizing intra-scan cycle and inter-scan cycle requirements of timing charts in LTL. The proposed formalization of LL programs and formal specification of timing charts provides a framework that readily enables the model checking of LL programs in SPIN, and sets the stage for automated repair of PLC programs as well as algorithmic incorporation of fault tolerance functionalities in PLC programs. Our approach can easily be extended to other PLC languages (e.g., ST) under the standard IEC 61131-3 [3]. Based on the proposed formal semantics, we also present transformation rules that enable automatic generation of Promela code from LL programs. To enable model checking in SPIN, one needs to specify program properties in LTL. We identify two types of properties for PLC programs due to their periodic nature, namely intracycle and intercycle (i.e., global) properties. . We show that LTL suffices for

capturing intracycle and intercycle properties specified in timing charts. Specifically, we define a precedence relation between the signal edges in a cycle and across multiple cycles. We then use Dwyer's specification patterns [11] to formally specify them in LTL. We also present a method for fault modeling and the design of fault tolerance properties in ladder logic and its Promela specifications. We demonstrate the contributions of this paper in the context of a carriage line system, where objects are moved by a carriage to a conveyor belt based on specific ordering and timing constraints.
Organization. Section 2 explains basics concepts of LL, timing charts and Promela. Section 3 defines a Promela semantics for LL programs. Then, Section 4 presents a method for characterizing and formalizing timing charts requirements. Section 5 studies the issue of modeling the environment of PLC programs. Subsequently, Section 6 presents a novel method for modeling faults as well as characterizing intracycle and intercycle fault tolerance. Finally, Section 7 makes concluding remarks and discusses future work.

2 Preliminaries

This section represents the basic concepts of the Ladder Logic programs (taken from [2]) in Subsection 2.1, and the timing charts in Subsection 2.2. Subsection 2.3 presents the syntax and semantics of the Promela modeling language [15].

2.1 Ladder Logic (LL) Programs

A PLC is composed of a microprocessor, relays, timers and counters. In an abstract sense, a PLC is a control ladder that comprises of (i) an input ladder which receives input signals from physical environment and provides them to the control logic; (ii) a control logic that processes input signals and determines the output values, and (iii) the output ladder which provides the generated output signals to the physical environment. The control logic is dictated by a PLC program specified in one of the programming languages under the IEC 61131-3 standard [3].
Carriage Line (CL) example. The Carriage Line (CL) system of Figure 1 (taken from [2]) comprises of a carriage, a conveyor belt and an arm that pushes goods to the conveyor belt. The CL system has a container from where goods are moved to the carriage. Then, the carriage moves forward until it reaches in front of the arm. The arm then pushes the object and then the carriage moves back and subsequently the arm pushes back too. Figure 1 illustrates the components and input-output signals of the CL system. The input signals start with an 'X' and the output signals start with 'Y'.

In LL, a program (see Figure 2) includes a set of rungs, where each rung may include a set of contacts, timers, counters and output devices from left to right. An execution of a ladder program starts from some initial input values (i.e., initial state) and goes through a sequence of time periods, called *scan cycles*, where in each scan cycle rungs are checked for execution in a top-down sequential fashion. Each rung conducts the input signals through its logic circuit in a left-to-right manner. The net result is that, in each scan cycle, the program scans the input ladder, executes the control logic of the rungs (top-down), and generates some outputs. At the start of a cycle, input signals are latched and any input signal change during the cycle will not take effect until the next cycle. Likewise, the generated output will actuate output devices at the end of each cycle.

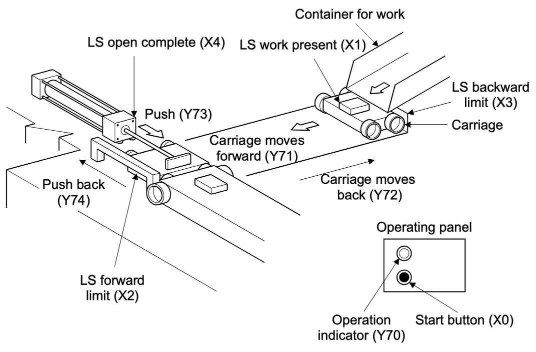

Figure 1: Carriage Line system taken from [2].

Figure 2 illustrates the LL program of the CL system. The first rung captures the logic that if the start button ($X0$) is pressed, and the completion flag $M2$ is false, then the operation indicator ($Y70$) must turn on and remain on as long as work is not completed yet; i.e., $M2$ is false. After $Y70$ output is activated, if the carriage is at the backward limit ($X3$) and the container has some work ($X1$), then a pulse $M1$ is

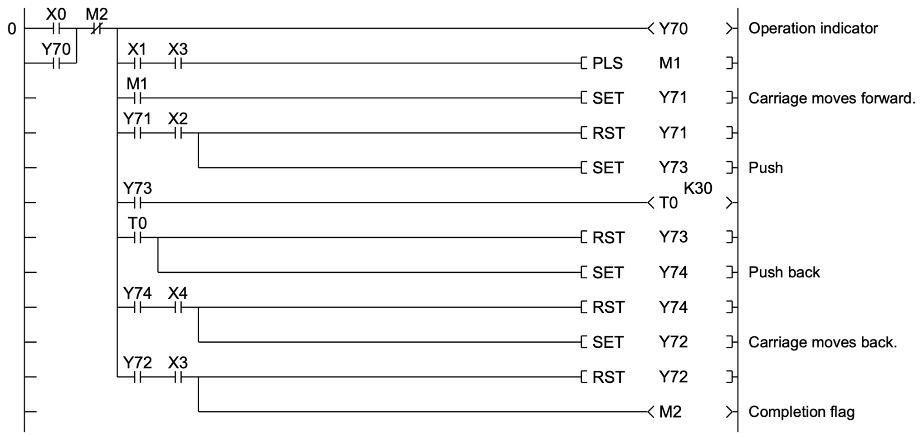

Figure 2: The ladder program for the Carriage Line system taken from [2].

generated, which remains on for one scan cycle. The pulse $M1$ activates the SET instruction, which sets $Y71$ to 1; i.e., the carriage starts moving forward. When the carriage reaches the forward limit ($X2$), the arm starts pushing ($Y73$) and $Y71$ is reset. The output $Y73$ triggers the timer $T0$ and $Y73$ remains on for 30 scan cycles. After that, the contact $T0$ closes, which in turn activates the push back signal $Y74$ of the arm. The actual push back starts when the arm is completely open ($X4$). When $Y74$ and $X4$ are on, the carriage move-back signal ($Y72$) turns on and $Y74$ is reset. When the carriage reaches the backward limit ($X3$), $Y72$ is reset and the completion flag $M2$ is set.

2.2 Timing Charts Requirements

The requirements of PLC programs are often specified as timing charts, especially in industry. As such, any method that is aimed at practical use by engineers must provide the means for systematic formalization of timing charts. A *timing chart* specifies how output signals are (de)activated depending on the changes in input signals, called *signal edges*. For example, in Figure 3 the *rising edge* of $X0$ represents that the start button is pressed and this change should result in a rising edge of $Y70$; i.e., indicator turns on.

One can say that the rising edge of $X0$ *precedes* the rising edge of $Y70$. Such precedence relations are required either in a scan cycle, called *intracycle* requirements/properties, or across several scan cycles, called *intercycle requirements*. For example, the precedence of $X0$ with respect to $Y70$ is intracycle, whereas the precedence of the rising edge of $X1$ and the rising edge of $X2$ is intercycle. Intuitively, once the work reaches the forward limit, i.e., rising edge of $X2$, that work has already been offloaded on the carriage from the container; i.e., rising edge of $X1$. Another class of requirements specified in timing charts includes *fixed-duration sequences of scan cycles* [6], which capture the delay periods of timers. For example, in the CL system, once a piece of work reaches the forward limit, it takes 3 seconds for the arm to push it to the conveyor belt. This period of 3 seconds contains 30 scan cycles for a scan cycle of 100msec.

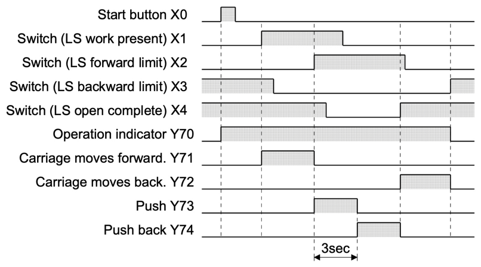

Figure 3: The timing chart of the Carriage Line system taken from [2].

A timing chart captures the requirements of an LL program in terms of a sequence of *events* and *observable states*. An *events* is either a change in some input signal or a time out generated by some timer. Events may cause a change in some output signals [6]. To model the rising and falling edges of signals, we consider a signal/variable V_o for every signal V, where V_o stores the value of V in the previous scan cycle. This way, we have a X_o (respectively, Y_o) signal for every input signal X (respectively, output signal Y). An *observable state* is a snapshot of the values of all input and output signals. The occurrence of each scan cycle may change the state of the program as it latches new inputs and sends out new outputs in one atomic step. Since the implementation of an LL program may contain internal devices (e.g., relays) or memory bits, the internal state of a program is actually a snapshot of all input-output signals and internal variables. For example, the program of Figure 2 includes two relays $M1$ and $M2$ as well as a timer, which are hidden from the viewpoint of an external observer. The *timing chart specification* of a program is in fact a sequence $s_0, e_0, s_1, e_1, \cdots$ of alternating events e_i and observable states s_i, for $i \geq 0$. As such, there is a one-to-one correspondence between each scan cycle and an observable state.

2.3 Promela and Linear Temporal Logic (LTL)

Process Meta Language (Promela) [1] is the modeling language of SPIN [15], which is a state-of-the-art model checker. The syntax of Promela is a variant of the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications. The semantics of Promela is based on an operational model that defines how the actions of processes are interleaved. An action (a.k.a *guarded command*) is of the form $grd \rightarrow stmt$, where the guard grd is a Boolean expression in terms of program variables and the statement $stmt$ updates program variables. When the guard grd holds (i.e., the action is *enabled*), the statement $stmt$ can be executed, which accordingly updates some variables. Actions can be atomic or non-atomic, where an atomic action (denoted by atomic {}) ensures that the guard evaluation and the execution of the statement is uninterrupted.

The requirements of a program are specified in terms of LTL expressions in the SPIN model checker [15]. The basic temporal operators include 'always', 'eventually', 'next' and 'until' respectively denoted by $\Box, \Diamond, X$ and U. Each temporal operator is defined over a sequence of states $\tau = s_0, s_1, \cdots$ and an LTL property Φ holds for τ *if and only if* (iff) Φ holds at s_0. For propositions ϕ and ψ (defined in terms of program variables) (i) $\Box\phi$ means that ϕ holds in all states of τ; (ii) $\Diamond\phi$ stipulates that ϕ holds in some state s_j of τ, for some $j \geq 0$; (iii) $X\phi$ states that ϕ holds in s_1, and (iv) $\psi U \phi$ holds at s_0 iff there is some state s_j where ϕ holds and ψ holds in all states from s_0 up to s_j, for $j \geq 0$.

3 Defining Promela Semantics of LL Programs

This section presents a method for formalizing LL programs in Promela. We first devise a Promela model that formalizes the cyclic/periodic nature of PLC programs (Subsection 3.1). Then, we present a method for transforming basic constructs of LL programs to Promela 3.2.

Challenges. Since our overarching goal is to devise a framework for control engineer where they can verify and synthesize fault tolerance in PLC programs, we consider a set of criteria for the intermediate formal semantics of LL programs.

- *Amenable to automated repair/redesign*: The incorporation of fault tolerance concern into an existing program is a special case of program repair where fault tolerance aspects must be captured while preserving functional concerns. As such, the formal semantics must be simple and have an efficient decision procedures.
- *Simplicity*: To the extent possible, we would like the formal semantics to benefit from a small syntactic footprint and a simple semantics that is easily understandable for control engineers.
- *Expressiveness for fault modeling*: The formal semantics of PLC programs should be sufficiently expressive for capturing different types of faults (e.g., soft errors, Byzantine attacks, crash faults, stuck-at faults)
- *Efficient decision procedures*: The language containment and non-emptiness problems must be decidable, and ideally solvable efficiently. This will help develop efficient verification and synthesis algorithms for LL programs.

- *Extensibility*: Since different vendors create their own off-shoots of PLC languages, the proposed formal semantics must be easily extensible to support such extensions. Thus, the formal semantics must have a core and extensions should easily be definable (preferably in terms of the linguistic constructs of the core).

3.1 Formalizing Cyclic PLC Executions

This section presents a Promela model that formalizes the execution semantics of cyclic PLC programs. Each PLC program may include several tasks with distinct priorities. In each scan cycle, the PLC code of all tasks are executed concurrently from higher priority to lower. A lower priority task cannot preempt a higher priority task. The code of each task may be an LL program (similar to Figure 2). If the conditions for the execution of a rung are not met, then that rung is skipped. Listing 1 presents a general semantics for the cyclic execution of PLC programs (regardless of the PLC language). For simplicity, Listing 1 has only two tasks, but it can easily be generalized to more tasks. The 'init' process (Lines 19-36) models the cyclic execution of PLC programs. Initially, we assume that tasks have just finished the previous cycle of execution, captured by the array cycleSync initialized to one in Line 4. Thus, the tasks will be initially waiting on Lines 8 and 14 until the 'init' process verifies that all tasks have finished the current cycle (Lines 25-29). Lines 30 and 31 must be implemented depending on the environment behaviors. Line 33 atomically resets the cycleSync array to indicate the start of a new scan cycle to all tasks.

Listing 1: The outline of a PLC program in Promela

```
 1  #define Low  1
 2  #define High 10
 3  #define N  2 // N captures the number of tasks in the PLC program
 4  bool cycleSync[N] = {1,1};
 5  bool  cntFlag;
 6
 7  proctype TaskOne() priority Low {
 8          Start: cycleSync[_pid -1] == 0;
 9          // The body of TaskOne comes here ...
10              cycleSync[_pid -1] = 1;
11          End: goto Start;
12  }
13  proctype TaskTwo() priority High {
14          Start: cycleSync[_pid -1] == 0;
15          // The body of TaskOne comes here ...
16             cycleSync[_pid -1] = 1;
17          End: goto Start;
18  }
19  init{
20    int i; // loop counter
21    atomic{  run TaskOne();    run TaskTwo();  }
22    cntFlag = 1;
23    startCycle:
24    // Wait for every task to finish this cycle.
25    L0: atomic{ for (i: 0 ..  N-1) {cntFlag = cycleSync[i] && cntFlag; } }
26          if
27          :: atomic{(cntFlag == 0) -> cntFlag = 1; goto L0;}
28          :: else skip;
29          fi;
30          // Update outputs (sent to the enviornment).
31          // Update inputs.  Models environment's behavior in updating
32          //  the input signals.
33          atomic{ for (i: 0 ..  N-1) {cycleSync[i] = 0; } }
34          // Start a new cycle. Let the tasks move on.
35    endCycle: goto startCycle;
36  } // end of init
```

3.2 From LL to Promela

This section illustrates how we can define a formal semantics in Promela for linguistic constructs of LL. The resulting Promela code of each task will be inserted in Lines 9 and 15 of Listing 1.

Coil output (OUT instruction). The first rung of the LL program in Figure 2 is an example of how an output coil (i.e., $Y70$) is energized/de-energized based on some input conditions (i.e., $(X0 \lor Y70) \land \neg M2$). For simplicity, consider a simple rung that connects the input signal $X0$ to the output coil $Y70$. Figure 4 illustrates the timing chart of the OUT instruction. Notice that the rising edge of $X0$ precedes the rising edge of $Y70$ in the same cycle. Once $X0$, which is a Normally Open (NO) contact, closes (i.e., $X0$ holds) the output coil $Y70$ energizes, and remains energized as long as $X0$ is closed. Once $X0$ is open again, the output coil $Y70$ is de-energized too.

Listing 2 presents the Promela code that we specify for the OUT instruction. Note that, there is no need to explicitly model the occurrence of edges because the new values of input signals are latched at the start of each cycle. Thus, if there is a change in the value of an input signal it will be detected in the start of the next cycle.

Pulse (PLS/PLF) instruction. Upon the rising/falling edge of an input signal, the pulse instruction creates a signal with the duration of a single scan cycle. Figures 5 and 6 respectively represent the LL construct and the timing chart of the PLS instruction upon the rising edge of an input signal $X0$. The relay $M5$ is turned on when the rising edge of $X0$ occurs, and it lasts for one scan cycle. Thus, $M5$ will pulse for one cycle every time there is a rising edge of $X0$. Symmetrically, $M0$ becomes 1 when there is a falling edge of $X1$ and it stays one for one scan cycle (see Figure 7). In the case of PLF, the precedence relation holds between the falling edge of $X1$ and the rising edge of $M0$.

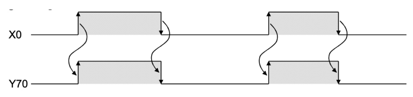

Figure 4: The timing chart of OUT (taken from [2]).

Listing 2: Promela code of the OUT instruction

```
atomic { if
               :: (X0_o == 0) && (X0_o == 1)    ->       Y70=1;
               :: (X0_o == 1) && (X0_o == 0)    ->       Y70=0;
         fi; }
```

Listing 3 models the PLS instruction in Promela. In addition to that, we reset $M5$ to zero at the end of the current cycle. That is, at the end of the proctype that contains the PLS instruction, we include $M5 = 0$ before going back to the start of a new cycle (e.g., before the ‘goto’ in Lines 11 and 17 of Listing 1). The Promela code of the PLF instruction is similar except that $M5$ is set to 1 when (X0_o == 1) && ((X0 == 0)) holds.

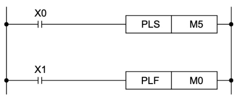

Figure 5: The PLS and PLF instruc-

SET/RST instructions. The SET instruction is to some extent similar to PLS except that when an output signal is set, it will remain on until it is reset back to 0. That is, its duration may last beyond a single scan cycle. Figure 8 illustrates the rungs corresponding to SET and RST commands. To understand the semantics of these instructions, we study their timing charts in Figure 9. Observe that, upon the rising edge of $X0$, the output $Y70$ turns on and stays on until $X1$ has a rising edge when the RST instruction is executed. We model SET in Listing 4. Note that, while this instruction is similar to PLS in terms of setting a signal/flag to one, it differs from PLS in that there is no need to reset that signal at the end of the cycle; i.e., the output signal remains on. The Promela code of RST is similar except that $Y70$ is set to zero on the falling edge of $X1$.

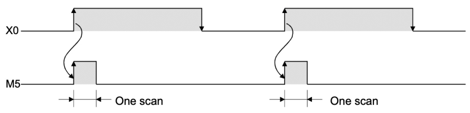

Figure 6: The timing chart of the pulse instruction on rising edge (PLS) (taken from [2]).

Timers. Timers form an important class of devices used in PLCs. There are different types of timers in PLCs but since measured timers represent the general behavior of a timer, we present only their formalization. Figure 10 illustrates the use of a timer with both Normally Open (NO) and Normally Closed (NC) contacts. To understand how the timer works, we use its timing chart in Figure 11. Upon the rising edge of $X5$, the coil of the timer $T0$ is energized and the timer is set to 30 as the number of scan cycles that the timer should delay until its NO contact $T0$ closes and turns the output $Y70$ on. In other words, after 30 scan cycles the timer times out. Moreover, when the timer coil is energized, it remains energized until a falling edge of $X5$ is observed, which in turn puts the contact $T0$ back to its NO state. The reverse occurs if the timer has a NC contact that turns the output signal $Y71$ off after 30 scan cycles (see Figure 11).

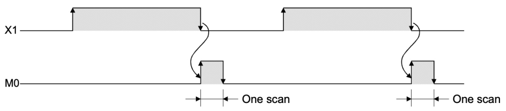

Figure 7: The timing chart of the pulse instruction on falling edge (PLF) [2].

Listing 3: Promela code of the PLS instruction

```
atomic { if
               ::(X0_o == 0) && (X0 == 1)       ->       M5=1;
               :: else skip;
```

```
            fi ; }
```

Listing 5 specifies the behavior of the timer in Promela. We first define the data type Timer that abstracts the timer, its contact and a Boolean flag indicating whether the timer has started working.

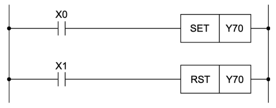

Figure 8: The SET and RST instructions [2].

The first atomic block of statements contains an 'if fi' statement whose first action sets the value of the timer to 30 upon observing the rising edge of $X5$. This action also makes sure that the NO contact remains open, and sets tmr.setFlag in order to indicate that the timer has started counting the scan cycles. The second action of 'if fi' statement just ensures that the NO contact remains open while the timer is counting down.

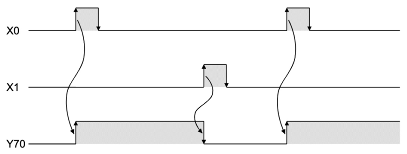

Figure 9: The timing chart of SET and RST instructions [2].

At the end of each scan cycle, tmr.value is decremented until it reaches zero. The second atomic block in Listing 5 implements this idea. Once the tmr.value becomes zero, the third action in the first atomic block resets tmr.setFlag, and closes the timer contact $T0$; i.e., sets tmr.contact. The timer coil remains energized until a falling edge of $X5$ is observed (the last action in the 'if fi' statement of the first atomic block), at which time the contact $T0$ becomes open (i.e., reset) and turns $Y70$ off. Likewise, the NC contact of $T0$ opens and turns $Y71$ on. The complete Promela code of the CL system is available in the Appendix.

Listing 4: Promela code of the SET instruction

```
atomic { if
              ::(X0_o == 0) && (X0 == 1)      ->      Y70=1;
              :: else skip;
         fi; }
```

4 Formalizing Timing Chart Properties

This section presents the formalization of precedence relations between signal edges in timing charts. Some of these precedence relations are required to occur in the same scan cycle, and some of them should occur globally over a sequence of scan cycles.

Intracycle Requirements. Inside a scan cycle, timing charts include rising/falling edges of signals coming one after another in a top down fashion. The top-down direction is imposed by the way rungs of a LL program are executed. Thus, we need to capture such precedence relations in terms of LTL expressions and make sure that they are required only in the span of a single cycle. More precisely, *between the start and end of each scan cycle, the occurrence of one rising/falling edge leads to the rising/falling edge of another signal.* To verify the precedence of signal edges (e.g., in SPIN), one has to specify them as LTL expressions. To this end, we find Dwyer's specification patterns [11] useful in showing that such properties can actually be specified in LTL, and there is no need for new variants of temporal logic (such as [12]). For example, the CL system requires that "with pressing the push button, the operation indicator will eventually turn on in the same cycle".

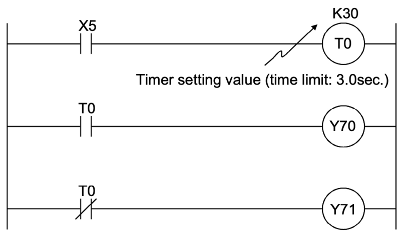

Figure 10: The rungs of a timer [2].

In the timing chart of Figure 3, this requirement is stated as *the rising edge of the input signal $X0$ must precede the rising edge of the output signal $Y70$ in the same scan cycle.* This requirement can be specified as the 'precedes' specification pattern "between Q and R we have P leadsto S", where Q and R are respectively substituted by the start and end of the cycle, and P and S respectively capture the rising edge of $X0$ and the rising edge of $Y70$. Formally, the precedes pattern is specified as $\Box((Q \wedge \neg R \wedge \Diamond R) \implies (P \implies (\neg R\ U (S \wedge \neg R)))U\ R)$. We then use the 'precedes' specification pattern and specify the LTL property X0Y70Edges in Lines 8 and 9 of Listing 6. We omit the specification of other intracycle requirements of Figure 3 because their specification is similar to that of X0Y70Edges. An example of such requirements includes the following: The falling edge of $Y73$ precedes the rising edge of $Y74$ (see Figure 3); i.e., after the arm pushes an object forward for 3 seconds, it will start pushing backward in the same cycle.

Intercycle Requirements. Our specification of the timing chart requirements that involve multiple cycles, i.e., *intercycle* or *global* requirements, is inspired by leadsto properties, where P 'leadsto' Q is specified as $\Box(P \implies \Diamond Q)$. For example, the CL system requires that "when there is a work present (rising edge of $X1$), it will eventually reach the forward limit (rising edge of $X2$) in subsequent cycles. Line 11 of Listing 6 formalizes this requirement. The X1LeadsToX2 property is slightly different from regular leadsto because the rising edge of $X2$ is expected to appear in subsequent cycles and not in the current cycle. That is why we have the 'Next' temporal operator (denoted by 'X') before '$\Diamond$riseEdgeX2'; i.e., in the next scan cycle, $\Diamond$riseEdgeX2 is satisfied.

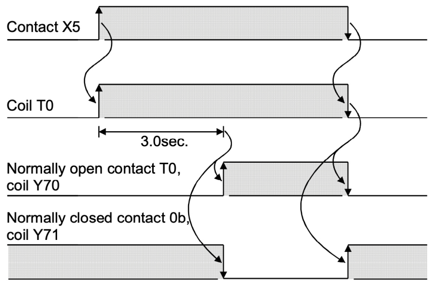
Contact X5
Coil T0
3.0sec.
Normally open contact T0, coil Y70
Normally closed contact 0b, coil Y71

Figure 11: The timing chart of a timer [2].

Listing 5: Promela code of timer

```
   typedef Timer {
      int value;
      bool setFlag;
      bool contact   };  // Define the Timer type.
   Timer tmr;  // Declare a timer.
 atomic { if
 ::((X5_o == 0) && (X5 == 1)) && (!tmr.setFlag) -> tmr.value = 30;
                                                    tmr.contact = 0;
                                                    tmr.setFlag = 1;
 ::((X5_o == 1) && (X5 == 1))  && (tmr.setFlag) && (tmr.value != 0)-> tmr.contact = 0;
 ::((X5_o == 1) && (X5 == 1))  && (tmr.setFlag) && (tmr.value == 0)-> tmr.contact = 1;
                                                                      tmr.setFlag = 0;
 ::((X5_o == 1) && (X5 == 0))&& (!tmr.setFlag)  -> tmr.contact = 0;
 :: else skip;
           fi;

atomic {  // This action must be executed at the end of the scan cycle.
if ::(tmr.value > 0) -> tmr.value--;
   ::else skip;
fi; // Decrease timer.
  }
```

Listing 6: Specification of timing chart properties in Promela.

```
1  #define start  (init@startCycle)
2  #define end    (init@endCycle)
3  #define riseEdgeX0   ((X0_o == 0) && (X0 == 1))
4  #define riseEdgeY70  ((Y70_o == 0) && (Y70 == 1))
5  #define riseEdgeX1   ((X1_o == 0) && (X1 == 1))
6  #define riseEdgeX2   ((X2_o == 0) && (X2 == 1))
7
8  ltl X0Y70Edges {[] ((start && !end && <>end) ->
9                        (riseEdgeX0 -> (!end U (riseEdgeY70 && !end))) U end) }
10
11 ltl X1LeadsToX2 {[] (riseEdgeX1 -> (X (<> riseEdgeX2))) }
```

5 Environment Model

This section presents a method for modeling the behavior of the environment of a PLC program. Specifically, we follow the timing chart in order to create such a model whose outputs are the input signals of the program. The inputs of the environment model includes the program's outputs and the value of a variable that counts the number of scan cycles. Thus, the process of creating a model for the environment has two inputs: a timing chart and a behavioral model that shows how the environment changes the input signals based on the output signals of a program. In the absence of a behavioral model, one can use just the timing chart in order to create a Promela model of the environment. Such a model changes the values of input signals depending on the current scan cycle only. For example, Listing 7 presents how we model the environment of the CL program based on the timing chart of Figure 3.

Listing 7: Promela model of the environment.

```
atomic{ if
            :: (scanCounter >= 1) && (scanCounter<2) -> X0 = 1;
            :: else X0 = 0;
        fi;     }
atomic{ if
            :: (scanCounter >= 3) && (scanCounter<7) -> X1 = 1;
            :: else X1 = 0;
        fi;    }
atomic{ if
            :: (scanCounter >= 5) && (scanCounter<13) -> X2 = 1;
            :: else X2 = 0;
        fi;    }
atomic{ if
            :: (scanCounter >= 0) && (scanCounter<6) -> X3 = 1;
            :: (scanCounter >= 6) && (scanCounter<20) -> X3 = 0;
            :: (scanCounter >= 20) && (scanCounter<=maxIter) -> X3 = 1;
            :: else skip;
        fi;   }
atomic{ if
            :: (scanCounter >= 0) && (scanCounter<8) -> X4 = 1;
            :: (scanCounter >= 8) && (scanCounter<11) -> X4 = 0;
            :: (scanCounter >= 11) && (scanCounter<=maxIter) -> X4 = 1;
            :: else skip;
        fi;    }
```

The code of Listing 7 is inserted in Line 31 of Listing 1. The 'scanCounter' is a global counter variable in the Promela model that starts from 0 and counts up to a maximum number of iterations, maxIter. Notice that, the initial state of the program is the valuation of all input-output signals right before the first vertical dashed line in Figure 3. As such, in the initial state, we have $X0 = 0, X1 = 0, X2 = 0, X3 = 1, X4 = 1$, and all output signals are zero. The first scan cycle starts with pressing the start button of the CL system, which turns $X0$ on for exactly on cycle. All the other inputs should remain unchanged. The first atomic action in Listing 7 captures this change. The remaining actions model how the other input signals change in the timing chart of Figure 3. Notice that, each action actually implements the time intervals when an input should be on/off. Moreover, the overlapping intervals are modeled by using appropriate range values in the guard conditions of the actions. For instance, there is an overlap between the interval where $X1$ is on and where $X2$ is on, however, $X2$ must turn on after $X1$ and $X1$ must turn off before $X2$. We have selected the current range values for a maxIter = 22 in order to expedite the simulation/verification of the Promela code of the CL program in SPIN. Nonetheless, as long as such 'happens-before' relations between the rising/falling edges of signals are preserved, the concrete lower and upper bounds of the ranges of the guards could be anything (given a specific maxIter). To determine whether the LL program of Figure 2 meets the requirements of the timing chart of Figure 3, we verify the generated Promela model against the precedence properties specified in Listing 6 using the environment model of Listing 7. The results of verification tell us whether with such changes in the input signals we get the correct timings in the output signals (as required by Figure 3). Our verification attempt confirms this. The Appendix presents the complete Promela model of CL and all the precedence properties that we verified.

6 Faults and Fault Tolerance

This section presents a method for modeling faults in Promela models of PLC programs in the context of intracycle and intercycle behaviors. We also define levels of fault tolerance with respect to intracycle and intercycle specifications. Avizienis *et al.* [5] classify faults based on the *conditions that cause them.* They define *soft (a.k.a. elusive) faults* (e.g., residual development faults) that cannot be easily reproduced due to complex conditions (i.e., combination of internal state and external conditions) under which they occur. By contrary, *hard faults* have activation conditions that are reproducible. We are mostly interested in the impact of faults on PLC programs instead of their reproducibility. As such, we consider *transient faults* that occur in a bounded amount of time and perturb the state of PLC programs without causing permanent damage. Such faults could occur due to a variety of reasons such as EMI and RF interference, hardware aging, etc. On the other hand, *permanent faults* persist over time and cause permanent failures such as failure of an input sensor, processor crash, etc. Next, we analyze the impact of transient and permanent faults in the context of cyclic PLC programs.

Intracycle and intercycle impact of faults. We call the faults that impact the input PLC signals *input faults.* That is, the input signals may be faulty in a transient or permanent way. In fact, we model input faults as part of environment's behavior. In terms of timing, input faults may be latched in the current or next cycle; i.e., input faults' impact may be delay for one scan cycle. Moreover, from the modeling point of view, input faults are synchronized with the scan cycle because any input change would be visible to the program at the start of the cycle. On the other hand, transient faults could impact the internal state of a PLC program at any moment regardless of scan cycle. That is, the impact of transient faults is asynchronous and does not follow the scan cycle timing. Thus, the transient fault model must be executed asynchronously with the PLC program tasks in Listing 1, whereas the input faults model is executed synchronously at the start of each scan cycle where inputs get updated. For example, if the start button in the CL system get stuck, then the input signal $X0$ becomes one forever; i.e., $X0 = 1$ holds always. This is a kind of faults, known as *stuck-at* that frequently occur in industrial environments.

We model stuck-at faults as illustrated in Listing 8, and insert its code in Line 31 of Figure 1, right after the Promela code that models the environment in Listing 7. The stuckatFlag is a Boolean flag that captures whether faults have occurred. The first 'if fi' statement has two actions none of which has a guard condition; i.e., SPIN non-deterministically picks one of them for execution, representing the non-deterministic occurrence of faults. Once faults occur and the start button gets stuck, it remains in that state until human operators repair it. Thus, from the point of view of program execution, the value of $X0$ is stuck at 1 forever. Notice that, since we insert the code in Listing 8 after the environment model, even if the value of $X0$ is reset by the environment model, the fault model sets it back to 1 before it is fed to the program.

Listing 8: Modeling Stuck-At faults.

```
bool stuckatFlag = 0;
atomic{   if
                :: X0 = 1; stuckatFlag = 1;
                :: skip;
                fi;  }
atomic {        if
                :: stuckatFlag == 1 -> X0 = 1;
                :: else skip;
                fi; }
```

We model transient faults as an independent active proctype that is run in parallel with program's model. For instance, transient faults may arbitrarily change the value of memory bits $M1$ and $M2$ in the CL system (see Figure 2) at any moment. Listing 9 presents how we model the impact of transient faults.

Listing 9: Modeling transient faults.

```
active proctype transientFaults(){
     if
          :: M1 = 0;
          :: M1 = 1;
     fi;
     if
        :: M2 = 0;
        :: M2 = 1;
     fi;
}
```

Levels of fault tolerance. Fault tolerance defines the degree up to which a program meets its safety and liveness specifications when faults occur [4]. Intuitively, when there is no faults, a fault-tolerant program satisfies its safety and liveness specifications. However, when faults occur, a *failsafe* fault-tolerant program satisfies its safety specifications only. Thus, a *failsafe* fault-tolerant program satisfies its safety specifications at all times. A *nonmasking* fault-tolerant program ensures that when faults occur it will eventually recover to states from where it meets its safety and liveness specifications. During such recovery, a nonmasking program may violate its safety, though. A *masking* fault-tolerant program is both failsafe and nonmasking; i.e., it satisfies its safety specifications always and ensures recovery to states from where both safety and liveness are met.

Intracycle fault tolerance. We now define the notion of fault tolerance with respect to the requirements of timing charts; i.e., precedence relations. As defined in Section 4, the precedence relations have a general template as follows *between the start and end of a scan cycle, an event P leadsto another event S*, where P and S denote the events of the rising/falling edge of some signals. Thus, the only way such an intracycle precedence relation may be violated in a scan cycle is that its premise (i.e., event P) holds, but its consequent (i.e., event S) does not. For example, in the LL program in Figure 2, a rising edge of $X0$ must be followed by a rising edge of $Y70$ in the same scan cycle, but if transient faults set $M2$ to one before $Y70$ is turned on by the first rung of the program, then the precedence property X0Y70Edges in Listing 6 is violated in the same cycle (because the rising edge of $Y70$ will never occur). *Can we detect and correct such failures in the same scan cycle; i.e., intracycle fault tolerance*? Since in each scan cycle rungs of a program are executed in a top-down fashion, any detection/correction must occur before program execution reaches the second rung. Otherwise, we have lost the chance of recovery in the current scan cycle. Thus, any fault tolerance mechanism must be included either in the first rung or right before the second rung. In this case, we need to detect the rising edge of $X0$ immediately followed by a rising edge of $M2$. If that occurs, then we must reset $M2$ to zero. More precisely, since $M2$ is a completion flag and must be set to one in the last rung of the program of Figure 2, $M2$ must be zero before and after the execution of every rung of the program (except the last one).

The dashed rungs in Figure 12 illustrate the fault tolerance functionality included for meeting the precedence property X0Y70Edges (in Listing 6) in the presence of transient faults that perturb $M2$. Notice that, the code added for fault tolerance is itself fault-tolerant for property X0Y70Edges. That is, even if faults perturb $M2$ while the fault tolerance rungs are executed, the rising edge of $Y70$ will occur eventually following the rising edge of $X0$. Listing 10 presents the corresponding Promela code of the dashed rungs in Figure 12. If $M2$ is further perturbed by transient faults after the code of Figure 12 is executed, we must ensure that $M2$ is recovered back to zero until we get to the last rung. For this purpose, $M2$ *must be reset to zero in every subsequent rung.* We have **used SPIN to verify the correctness** of the resulting program, which is a *masking* fault-tolerant program for X0Y70Edges against faults that perturb $M2$.

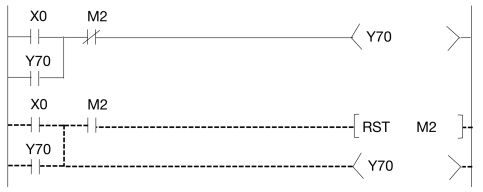

Figure 12: The LL code added for tolerating transient faults that perturb $M2$.

Listing 10: Tolerating transient faults that perturb $M2$.

```
atomic{    if
        :: (X0 || Y70) && M2 -> M2 =0;
        :: else skip;
           fi;
           if
        :: (X0 || Y70)  -> Y70 = 1;
        :: else skip;
           fi;
}
```

Intercycle fault tolerance. Precedence relations that occur over multiple cycles are examples of bounded liveness properties. For example, in the CL system we have the following requirement: *When there is a work present (i.e., rising edge of $X1$ occurs), it will reach the forward limit in exactly 30 cycles (i.e., rising edge of $X2$ will happen in exactly 30 cycles).* The violation of this property could be due to two types of faults: physical and cyber. First, if the carriage is stuck in the middle of its way to the forward limit (for mechanical reasons), then the rising edge of the signal $X2$ will never be observed. Second, the carriage does actually reach the forward limit but the signal $X2$ does not turn on due to some noise/fault. In both cases, any fault tolerance scheme can raise a flag for the operator. While the detection of this type of faults can be done in software, their correction may need human intervention in the physical space. Moreover, faults may affect timers and lead to the violation of timing constraints. For instance, in the aforementioned requirement of the CL system, the rising edge of $X2$ may occur in more or less number of scan cycles than 30 cycles. Designing failsafe fault tolerance against such types of timing faults is impossible because faults directly violate safety properties. Nonetheless, to contain the impact of faults, it is desirable to detect them and implement fault containment schemes. This problem is part of our ongoing investigations.

7 Conclusions and Future Work

This paper presented a novel approach for (i) formalizing Ladder Logic (LL) programs in Promela; (ii) characterizing timing charts requirements as precedence relations between signal edges, and formalizing them in LTL, and (iii) modeling faults and designing fault tolerance for LL programs. The characterization of timing charts requirements as intracycle and intercycle precedence relations leads to defining intracycle and intercycle fault tolerance for cyclic PLC programs, in general. To the best of our knowledge, this is the first work that studies fault tolerance in the context of cyclic PLC programs, especially for ladder logic. We are currently working on several extensions of this work, including (1) automation of the formalization method, both for LL programs and for timing charts requirements; (2) algorithmic incorporation of fault tolerance functionalities in LL programs (while benefiting from our past work on synthesis of fault-tolerant concurrent programs [16, 17]), and (3) mechanical proof of correctness of the formalization approach.

References

[1] Promela language reference. `http://spinroot.com/spin/Man/promela.html`.

[2] Mitsubishi PLC Training Manual: Basic Course for GX Works2, 2012. `https://dl.mitsubishielectric.com/dl/fa/document/manual/school\_text/sh081123eng/sh081123enga.pdf`.

[3] International Electrotechnical Commission, IEC International Standard IEC 61131-3 (Programming Languages), 2013. `https://www.plcopen.org/iec-61131-3`.

[4] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] C. Belo Lourenço, D. Cousineau, F. Faissole, C. Marché, D. Mentré, and H. Inoue. Automated formal analysis of temporal properties of ladder programs. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2022.

[7] E. Brinksma, A. Mader, and A. Fehnker. Verification and optimization of a PLC control schedule. *International Journal on Software Tools for Technology Transfer*, 4:21–33, 2002.

[8] L. A. Bryan and E. A. Bryan. *Programmable controllers: Theory and implementation*. Industrial Text Company, 1997.

[9] D. Darvas, E. Blanco Vinuela, and B. Fernández Adiego. Plcverif: A tool to verify plc programs based on model checking techniques. 2015.

[10] D. Darvas, I. Majzik, and E. B. Viñuela. Conformance checking for programmable logic controller programs and specifications. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2016.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, 1998.

[12] N. O. Garanina, I. S. Anureev, V. E. Zyubin, S. M. Staroletov, T. V. Liakh, A. S. Rozov, and S. P. Gorlatch. A temporal logic for programmable logic controllers. *Automatic Control and Computer Sciences*, 55(7):763–775, 2021.

[13] L. Garcia, S. Mitsch, and A. Platzer. HyPLC: Hybrid programmable logic controller program translation for verification. In *Proceedings of the 10th acm/ieee international conference on cyber-physical systems*, pages 47–56, 2019.

[14] I. Grobelna, M. Grobelny, and M. Adamski. Petri nets and activity diagrams in logic controller specification-transformation and verification. In *Proceedings of the 17th International Conference Mixed Design of Integrated Circuits and Systems-MIXDES 2010*, pages 607–612. IEEE, 2010.

[15] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[16] A. Klinkhamer and A. Ebnenasir. A software framework for automated synthesis of self-stabilization. `http://asd.cs.mtu.edu/projects/protocon/`.

[17] A. Klinkhamer and A. Ebnenasir. Shadow/puppet synthesis: A stepwise method for the design of self-stabilization. *IEEE Trans. Parallel Distrib. Syst.*, 27(11):3338–3350, 2016.

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

[19] O. Ljungkrantz, K. Akesson, M. Fabian, and C. Yuan. Formal specification and verification of industrial control logic components. *IEEE Transactions on Automation Science and Engineering*, 7(3):538–548, 2009.

[20] I. D. Lopez-Miguel, J.-C. Tournier, and B. F. Adiego. Plcverif: Status of a formal verification tool for programmable logic controller. *arXiv preprint arXiv:2203.17253*, 2022.

[21] J. Luo, Q. Zhang, X. Chen, and M. Zhou. Modeling and race detection of ladder diagrams via ordinary petri nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(7):1166–1176, 2017.

[22] X. Mao, Y. Zhang, J. Shi, Y. Huang, and Q. Li. A refinement development approach for enhancing the safety of plc programs with event-b. *Science of Computer Programming*, 215:102763, 2022.

[23] K. L. McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

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

[25] R. Wiśniewski, G. Bazydło, P. Szcześniak, and M. Wojnakowski. Petri net-based specification of cyber-physical systems oriented to control direct matrix converters with space vector modulation. *IEEE Access*, 7:23407–23420, 2019.

Appendix: Complete Promela Model of the Fault-Tolerant Carriage Line System

```
#define maxIter 21
#define start (Carriage@StartCycle)
#define end   (Carriage@EndCycle)

// Intra cycle properties:  For example, "between_Q_and_R_we_have_P_leadsto_S"
//               [](((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)
// Q can be the start of the cycle and R can denote the end of a cycle

 #define posEdgeX0  ((X0_o == 0) && (X0 == 1))
 #define posEdgeY70 ((Y70_o == 0) && (Y70 == 1))

 #define posEdgeX1  ((X1_o == 0) && (X1 == 1))
 #define posEdgeY71 ((Y71_o == 0) && (Y71 == 1))

 #define posEdgeX2  ((X2_o == 0) && (X2 == 1))
 #define posEdgeY73 ((Y73_o == 0) && (Y73 == 1))

 #define negEdgeY73 ((Y73_o == 1) && (Y73 == 0))
 #define posEdgeY74 ((Y74_o == 0) && (Y74 == 1))

// With pressing the push button, the operation indicator will eventually turn on in the same cycle.
ltl X0Y70Edges {[] ((start && !end && <>end) -> (posEdgeX0 -> (!end U (posEdgeY70 && !end))) U end) }

// When there is a work present, the carriage eventually moves it forward in the same cycle.
ltl X1Y71Edges {[] ((start && !end && <>end) -> (posEdgeX1 -> (!end U (posEdgeY71 && !end))) U end) }

// When there is a work present (positive edge of X1), it will eventually reach the forward limit
// (positive edge X2) in subsequent cycles
ltl X1LeadsToX2 {[] (posEdgeX1 -> X (<> posEdgeX2)) }

// When the work is stopped at the forward limit (positive edge of X2), then the arm will eventually start pushing
//  it to the other conveyer belt (positive edge of Y73) in the same cycle.
ltl X2Y73Edges {[] ((start && !end && <>end) -> (posEdgeX2 -> (!end U (posEdgeY73 && !end))) U end) }

// The push forward takes 3 seconds (i.e., 30 cycles). That is, positive edge of Y73 will be followed
// by a negative edge of Y73 in 30 scan cycles.

// After the arm pushes the work forward for 3 seconds (falling edge of Y73), it will eventually start pushing
// backward in the same cycle (rising edge of Y74).
ltl Y73Y74Edges {[] ((start && !end && <>end) -> (negEdgeY73 -> (!end U (posEdgeY74 && !end))) U end) }

bool X0, X1, X2, X3=1, X4=1;
bool Y70, Y71, Y72, Y73, Y74;

bool X0_o, X1_o, X2_o, X3_o=1, X4_o=1;  // Capture the values of input/output signals in the previous scan cycle.
bool Y70_o, Y71_o, Y72_o, Y73_o, Y74_o;
int scanCounter=1;

typedef Timer {
      int value;
      bool setFlag;
      bool contact   };

    bool M1; // This is an internal variable of Carriage() but we have moved it here to
             //  model the impact of faults on it.
    bool M2=0;
  active proctype Carriage() {


  bool precond;


  Timer tmr;
  tmr.setFlag = 0; // Timer has not energized yet
  tmr.contact = 0; // Timer has a normally open contact

 // Setting the input values
atomic{ if
            :: (scanCounter >= 1) && (scanCounter<2) -> X0 = 1;
            :: else X0 = 0;
        fi;
}

 atomic{ if
            :: (scanCounter >= 3) && (scanCounter<7) -> X1 = 1;
            :: else X1 = 0;
        fi;
 }
 atomic{
        if
            :: (scanCounter >= 5) && (scanCounter<13) -> X2 = 1;
            :: else X2 = 0;
        fi;
 }
 atomic{
        if
            :: (scanCounter >= 0) && (scanCounter<6) -> X3 = 1;
            :: (scanCounter >= 6) && (scanCounter<20) -> X3 = 0;
            :: (scanCounter >= 20) && (scanCounter<=maxIter) -> X3 = 1;
            :: else skip;
        fi;
 }
 atomic{
        if
            :: (scanCounter >= 0) && (scanCounter<8) -> X4 = 1;
```

```
             :: (scanCounter >= 8) && (scanCounter<11) -> X4 = 0;
             :: (scanCounter >= 11) && (scanCounter<=maxIter) -> X4 = 1;
             :: else skip;
        fi;
 }
 StartCycle:
  atomic { precond = (X0 || Y70) && !M2; }
  atomic {
       if
                 ::  precond  -> Y70 = 1; // Energizing. This is an OUT instruction on device Y70.
                 ::  !precond  -> Y70 = 0; // Denergizing.
                                          // When the condition becomes false, the device should turn off.
             fi;
                  }


 atomic{  // Added for recovery against faults that perturb M2
     if
         :: (X0 || Y70) && M2 -> M2 =0;
         :: else skip;
            fi;
     if
         :: (X0 || Y70)  -> Y70 = 1;
         :: else skip;
            fi;
 }


  atomic { if
                 ::  precond && X1 && X3 -> M1 = 1;
                 :: else skip;
             fi;  }    // Pulse M1
  atomic { if
                 ::   precond && M1 -> Y71 = 1;
                      :: else skip;
             fi;  }  // Set Y71. The SET instruction sets a device once a condition becomes true.
                  // The device remains on even if the condition is fallsified. A Reset inst can turn the device off.

  atomic { if
                 ::  precond && Y71 && X2 -> Y71 = 0;  Y73 = 1;
                           :: else skip;
             fi; } // Reset Y71 and  Set Y73
  atomic { if
                 ::  precond && Y73 && (!tmr.setFlag) -> tmr.value = 3; tmr.contact = 0; tmr.setFlag = 1;
                 ::  precond && Y73 && (tmr.setFlag) && (tmr.value != 0) -> tmr.contact = 0;
                 ::  precond && Y73 && (tmr.setFlag) && (tmr.value == 0) -> tmr.contact = 1;  tmr.setFlag = 0;
                 ::  !(precond && Y73) -> tmr.contact = 0; // Once the enabling condition is no longer true,
                                                         // the timer contact must go back to its normal condition.
                 :: else skip;
             fi;
  } // Set the timer. timer.contact plays the role of contact T0 in the ladder program


  atomic { if
                 ::  precond && (tmr.contact == 1) -> Y73 = 0;  Y74 = 1;
       :: else skip;
             fi; } // Reset Y73 and set Y74 when timer goes on; i.e., 30 scan cycles have past.

  atomic { if
                 ::  precond && Y74 && X4 -> Y74 = 0; Y72 = 1;
                 :: else skip;
             fi;  } // Reset Y74 and set Y72

  atomic { if
                 :: precond && Y72 && X3 -> Y72 = 0; M2 = 1;
                  :: !(precond && Y72 && X3) ->  M2 = 0;
             fi;   } // Reset Y72 and set the completion flag M2





  // M1 must be reset at the end of the cycle because it is supposed to be a pulse.
   atomic {  // end of cycle activities
    M1 =0;
    if  :: (tmr.value > 0) -> tmr.value--;
        :: else skip;
    fi; // Decrease timer.


   }

    if
    :: (scanCounter <= maxIter +1) -> scanCounter++;
    :: else goto exit;
    fi;

EndCycle:
// This code is included for specifying the timing chart requirements.
 X0_o = X0; X1_o = X1;  X2_o = X2; X3_o= X3; X4_o = X4;
 Y70_o = Y70; Y71_o = Y71; Y72_o = Y72; Y73_o = Y73; Y74_o = Y74;



    goto StartCycle;

    exit: skip;
}

active proctype faults(){

    // If the X1 sensor becomes faulty then the property X1Y71Edges is violated
    // because X1 may become 1 and Y71 does not become one in the same cycle.
    // This is sort of a safety property.
    // We can recover from this but no failsafe program exists.
    // Recover if we have not seen a falling edge of X0 yet.
    // !negEdgeX0 && (X1 == 1) -> X1 = 0;


    if
        :: M2 = 0;
        :: M2 = 1;
    fi;
}
```