Automated Verification of Temporal Properties of Ladder Programs

Cláudio Lourenço, Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, Hiroaki Inoue

Automated Verification of Temporal Properties of Ladder Programs *

Cláudio Belo Lourenço[1], Denis Cousineau[2], Florian Faissole[2], Claude Marché[1], David Mentré[2], and Hiroaki Inoue[3]

[1] Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France
[2] Mitsubishi Electric R&D Centre Europe, Rennes, France
[3] Mitsubishi Electric Corporation, Amagasaki, Japan

Abstract. Programmable Logic Controllers (PLCs) are industrial digital computers used as automation controllers in manufacturing processes. The Ladder language is a programming language used to develop PLC software. Our aim is to prove that a given Ladder program conforms to an expected temporal behaviour given as a *timing chart*, describing scenarios of execution. We translate the Ladder code and the timing chart into a program for the Why3 environment, within which the verification proceeds by generating *verification conditions*, to be checked valid using automated theorem provers. The ultimate goal is two-fold: first, by obtaining a complete proof, we can verify the conformance of the Ladder code with respect to the timing chart with a high degree of confidence. Second, when the proof is not fully completed, we obtain a *counterexample*, illustrating a possible execution scenario of the Ladder code which does not conform to the timing chart.

Keywords: Ladder language for programming PLCs, Timing charts, Formal specification, Deductive verification, Why3 environment.

1 Introduction

Programmable Logic Controllers (PLCs) are industrial digital computers used as automation controllers in manufacturing processes, such as assembly lines or robotic devices. PLCs can simulate the hard-wired relays, timers and sequencers they have replaced, via software that expresses the computation of outputs from the values of inputs and internal memory. The Ladder language, also known as Ladder Logic, is a programming language used to develop PLC software. This language uses circuit diagrams of relay logic hardware to represent a PLC program by a graphical diagram. This language was one of the first available for programming PLCs, and is now standardised in the IEC 61131-3 standard [17]. It is one language among other languages for programming PLCs, and is still widely used and very popular among technicians and electrical engineers.

* This work has been partially supported by the bilateral contract ProofInUse-MERCE between Inria team Toccata and Mitsubishi Electric R&D Centre Europe, Rennes.

Because of the widespread usage of PLCs in industry, verifying that a given Ladder program conforms to its expected behaviour is of critical importance. In this work, we consider the description of the expected temporal behaviour under the form of a *timing chart*, describing scenarios of execution. Our approach consists in automatically translating the Ladder code and the timing chart into a program written in the WhyML language, which is the input language of the generic Why3 environment for deductive program verification [6]. In WhyML, expected behaviours of program are expressed using *contracts*, which are annotations expressed in formal logic. The Why3 environment offers tools for checking that the WhyML code conforms to these formal contracts. This verification process is performed using automated theorem provers, so that at the end, if the back-end proof process succeeds, the conformance of the Ladder code with respect to the timing chart is verified with a high degree of confidence. Yet, a complete formal proof is not the only expected feedback from our tool chain: we also want to obtain useful feedback when the proof does not succeed, our long-term goal being to build a tool that would be useful to regular Ladder programmers. More precisely, in such a case of proof failure, we aim at obtaining a *counterexample* which must illustrate a possible execution scenario of the Ladder code which does not conform to the timing chart.

This paper is organised as follows. We start in Section 2 by introducing the basics of Ladder programming, and the way their expected temporal behaviours are expressed using timing charts. The translation of Ladder code and timing charts into WhyML programs is described in Section 3. Section 4 presents our experiments and their results, both in the case of a complete proof success and in the case of a proof failure, where a counterexample is generated. We discuss related work and future work in Section 5. For sake of concision some technical details are omitted, such details are available in an extended research report [4].

2 Introduction to Ladder Programming

A Ladder program (a *diagram)* takes inputs values (*contacts*) that correspond to the fact that physical relays are either wired, not wired, pulsing (rising edge) or downing (falling edge), and other values stored in the internal memory of the PLC (Boolean values, integers, floating-point values, strings, *etc.*). A Ladder program can output Boolean values to the physical relays of the factory (*coils*) or it can call instructions, that may modify the values of the internal memory of the PLC (*devices*). Graphically, contacts are located at the left of the diagram. They can be combined in a serial way (Boolean conjunction) or in a parallel way (disjunction). Coils and instructions are activated when the combination of contacts at their left gives a wired value, and they can also be parallelised (in that case, they are either all activated or all deactivated). A line with contacts, coils and instructions is called a *rung*, and a program is composed of several rungs. Such a Ladder program is executed cyclically in a synchronous way: first inputs are read, then the program is executed and eventually outputs are written. One single execution of the program is called a *scan*.

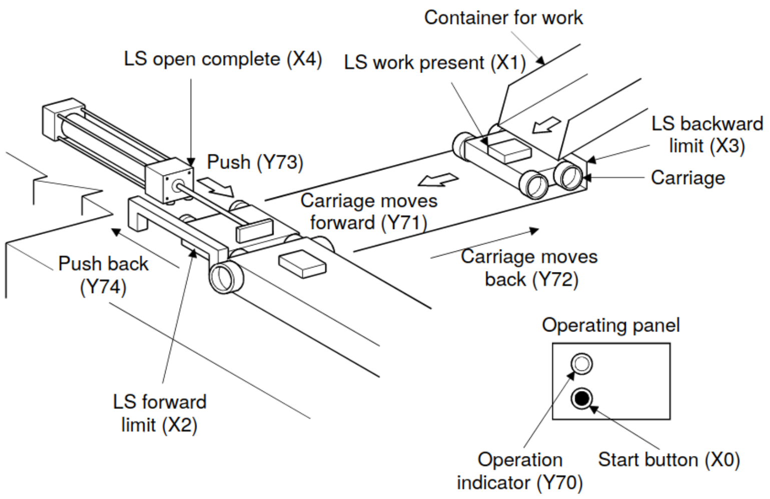


Fig. 1: Carriage line control: System description

Running example. A rather simple example of a PLC controlling a carriage line is depicted in Figure 1, with the corresponding Ladder program in Figure 2. This example comes from a Mitsubishi Electric training manual for programming PLCs [14]. To our knowledge, timing charts are generally used to specify programs of comparable size, *e.g. Function Blocks*, which are kind of library functions that are shipped together with a PLC and a programming environment. We illustrate some principles of Ladder Logic on the first rung of this example: this rung expresses the fact that output Y70 receives the value of the Boolean formula (X0 $\vee$ Y70) $\wedge$ ($\neg$ M2), *i.e.* if the corresponding physical devices are activated such that the Boolean formula is true, then Y70 is activated, and is deactivated otherwise.

The program also makes use of the Ladder instructions SET, RST and PLS. The SET instruction activates its device argument (either an internal memory device or an output device) when its input is activated, and does nothing otherwise. For instance, in Figure 2, Y71 is activated when both the common front (X0 $\vee$ Y70) $\wedge$ ($\neg$ M2) and the internal memory device M1 are activated. The RST instruction is the opposite: the device argument is deactivated when the input is activated. The PLS (pulse) instruction activates its device argument on a rising edge of its input, *i.e.* when the input has just been activated, then it deactivates its device argument on the next scan.

The diagram also uses a timer instruction on a special device T0 which is activated once the timer finishes. When its input is activated, the instruction sets the threshold (here 30) of the timer and increments a counter. After 30 consecutive scans in which both the common front and output Y73 are activated,

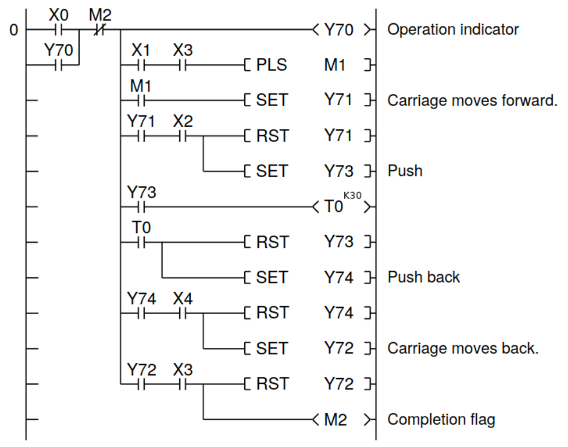


Fig. 2: Carriage line control: Ladder program

the device T0 is activated (and it remains activated until the input of the timer instruction is deactivated).

Specification of expected temporal behaviour. Because of its synchronous nature, the language hardly lends itself to exhaustive functional specifications. Since the work made on AutomationML [11] by an industrial and academic consortium, the practice, among PLC designers, is to use the *timing chart* paradigm, which describes the expected temporal behaviour of the PLC for a nominal execution scenario. A timing chart specifies the evolution of outputs over the execution of scans, according to the evolution of inputs. It is made of a succession of *events*, *i.e.* scans with either changes of inputs that may lead to changes of outputs, or endings of timers that lead to changes of outputs. Events are separated by *stable states*, *i.e.* arbitrary-length successions of scans in which the values of both inputs and outputs are unchanged.

Figure 3 depicts the timing chart specification of the carriage line control example. Events of the timing chart are depicted as $\sharp 1, \sharp 2, \ldots, \sharp 11$. In the rest of the paper, we use the notation $\sharp(1 \hookrightarrow 2), \ldots, \sharp(10 \hookrightarrow 11)$ to depict stable states. The initial and final states of the timing chart are respectively depicted by $\sharp idle$ and $\sharp end$. The timing chart of Figure 3 also contains a fixed-duration sequence of events and stable states (represented by an arrow, between events $\sharp 5$ and $\sharp 8$), whose duration is 3 seconds. We call *fixed-duration sequence* the concerned sequence of events and stable states. Typically, the Ladder program is executed periodically every 100 milliseconds, therefore, the fixed-time period of 3 seconds is made of 30 scans. Here, the given implementation uses the timer device T0 in order to satisfy this aspect of the specification.

Our main goal is the verification that a Ladder diagram conforms to such a timing chart specification. A first idea would be to envision the use of deductive verification techniques, in the wake of our previous work on Ladder instruction-

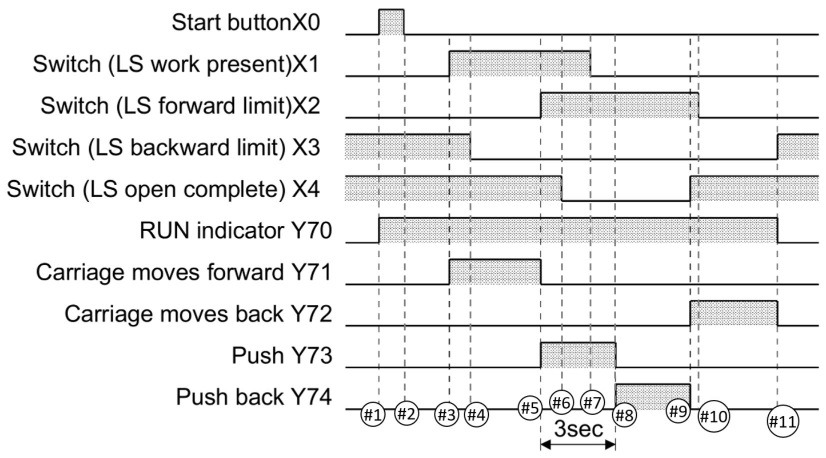


Fig. 3: Timing chart specification for the Carriage line control

level verification [7]. However, not all variables used in the Ladder program of Figure 2 are addressed by the timing chart. Indeed, internal memory devices (*e.g.* M1 and M2) and timers (*e.g.* T0) are introduced by the developer in order to make the program satisfy its specification, but do not belong to this specification. As an example, in the carriage line control program, the M2 device acts as a termination flag which stops the execution of the PLC as soon as it is activated. There is no doubt that M2 remains false during execution of the timing chart scenario. However, deductive verification would lack this information to check that outputs satisfy their specification. This kind of issue is at the heart of our strategy that is to integrate a method for *inferring loop invariants*.

3 Translation of Ladder Programs to WhyML

Our prototype automatically translates Ladder programs given as XML files and timing charts given as PlantUML [18] files into WhyML programs. After a short introduction to the Why3 environment in Section 3.1, we describe in Section 3.2 how we translate the Ladder program itself, and in Section 3.3 how we use this translation for modeling the successive executions of the program and verifying that it satisfies the given timing chart.

3.1 The Why3 Environment

Why3 is an environment for deductive program verification, providing the language WhyML for specification and programming [6]. A detailed introduction to Why3 is given in our extended report [4]. Among the recent features of Why3 of particular interest for our work are the ability to *generate loop invariants* and to *produce counterexamples* when a proof fails [8]. Indeed, the first of these features

```
val b : ref bool
val x : ref int

let toy () : unit
  requires { 0 <= !x <= 10 }
  writes { b, x }
  ensures { not !b }
  ensures { !x <= 200 }
= b := false;
  while (!x < 100) do
    b := (!x < 50);
    if !b then x := !x + 2
          else x := !x + 3;
  done;
  assert { !x >= 75 }
```

(a) Toy example of WhyML code

```
let set (input : bool)
        (device : ref bool) : unit
 writes  { device }
 ensures { !device ↔
           (input ∨ old !device) }
= if input then device := true
```

(b) the SET instruction in WhyML

Fig. 4: Examples of WhyML code

had to be improved in order to support our work on Ladder programs, this is a contribution that some of us made to Why3 [4].

We illustrate those features on a toy WhyML program presented in Figure 4a. This code involves two global variables, `b` of type Boolean and `x` of type integer (a mathematical, unbounded integer in WhyML). The function `toy` takes no arguments, and is equipped with a formal contract involving a pre-condition (keyword `requires`) stating that the value of `x` on function entry is required to lie between 0 and 10, and two post-conditions (keyword `ensures`) stating respectively that at exit, `b` is false and that `x` is smaller than 200. The clause `writes` expresses which global variables are potentially modified by that function. Notice the WhyML syntax for mutable variables, inspired by ML, requiring to write an exclamation mark to access their values. The body of that function is a simple imperative code involving a while loop and a conditional. This code ends by an other kind of formal annotation, namely a code assertion stating that the value of `x` must be greater or equal to 75 after the loop.

Given such an annotated code, the Why3 core engine generates three *verification conditions* (VCs), corresponding to the assertion and the two post-conditions. When calling provers for attempting to prove these VCs, only the assertion is proved valid: it directly follows from the negation of the loop condition. None of the post-conditions are proved valid, which is expected in the classical setting of deductive verification, because for proving properties about loops one should state appropriate *loop invariants*. These could be added by hand, but to make the process more automatic we rely on the automatic generation of such invariants. We use a technique based on *abstract interpretation*, for which an early prototype existed for Why3 [1], prototype that we extended in particular to support Boolean variables [4]. The generated loop invariant is then as follows:

```
(!b = false ∧ 0 <= !x <= 10) ∨ (!b = true  ∧ 2 <= !x <= 51) ∨
(!b = false ∧ 53 <= !x <= 102)
```

and with this loop invariant, the post-conditions are proved valid.

Assume now that we replace the loop condition with `(!x < 300)`. Still assuming that we ask for generation of a loop invariant, all generated VCs are proved except the second post-condition. For this VC, Why3 proposes a *counterexample* where the values of `b` and `x` at loop exit are respectively `false` and 300. Indeed, these values satisfy the loop invariant, but with those the post-condition `!x <= 200` is not valid.

3.2 Translation of Ladder Codes

The translation relies on models of Ladder instructions as WhyML functions, defined by some of the authors in a previous work [7]. For example, Figure 4b depicts the function that corresponds to the SET instruction. This function takes two arguments, first the input of the instruction (whether it should be activated or not), and second the device on which it may have an effect. Both the code and the contract of the function state the intended behaviour of the SET function: if the instruction is activated then the considered device is activated (otherwise its value does not change). The WhyML functions modeling RST, PLS, and timer instructions are detailed in the extended research report [4].

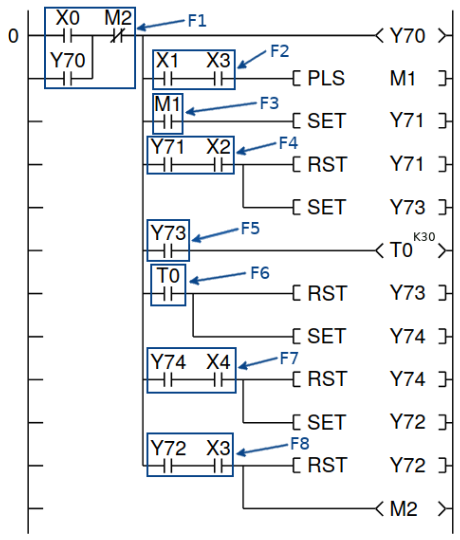
Given this formalisation of Ladder instructions, we can now give, in Figure 5b, the translation of the full Ladder program of Figure 2. The translation makes use of auxiliary variables `f1`,. . . ,`f8` which corresponds to the common fronts depicted on Figure 5a.

3.3 The Ladder loop, and the Encoding of Timing Charts

By definition, timing charts are made of successive events and stable states. Checking that a program conforms to a timing chart means that, under the hypotheses on input values, the values of outputs are correct according to the order of appearance of events and stable states in the timing chart scenario. In addition, fixed-time duration information (timer-related sequence of events) also need to be verified. We propose and implement an automatic process that takes a Ladder diagram and a timing chart specification and returns the corresponding WhyML formalisation.

Events and stable states as loops. The formalisation is made of a succession of *do-while* style loops[4] (except for the initial stable state of the timing chart). The

[4] There are no *do-while* loops in WhyML, we just mean by *do-while* style loop a code piece of the following form with two occurrences of the loop body: "`body; while cond do body done`"



```
let f1 = (!x0 || !y70)
         && (not !m2) in
y70 := f1;
let f2 = !x1 && !x3 in
pls (f1 && f2) m1 cc0;
let f3 = !m1 in
set (f1 && f3) y71;
let f4 = !y71 && !x2 in
rst (f1 && f4) y71;
set (f1 && f4) y73;
let f5 = !y73 in
timer_coil (f1 && f5) t0 30;
let f6 = timer_contact t0 in
rst (f1 && f6) y73;
set (f1 && f6) y74;
let f7 = !y74 && !x4 in
rst (f1 && f7) y74;
set (f1 && f7) y72;
let f8 = !y72 && !x3 in
rst (f1 && f8) y72;
m2 := f1 && f8
```

(a) Ladder code with common fronts (b) WhyML encoding

Fig. 5: Encoding of one scan of the Ladder program for the carriage line control

body of each loop corresponds to the WhyML formalisation of one scan of the Ladder program. Each do-while loop corresponds to a pair made of an event (the first iteration *do*) and the following stable state (*while*). The guard of the loop corresponds to the assumptions on inputs, *i.e.* the values taken by the inputs at the corresponding event and during the following stable states. The verification conditions on outputs are modelled as loop invariants: the invariant initialisation corresponds to the event while its preservation corresponds to the stable state.

The initial state of the timing chart (values of devices before the PLC starts) is handled in its own way. Basically, all outputs and internal memory devices are initially deactivated. The initial values of inputs are read at the beginning of the timing chart. The initial state is formalised as a while loop (and not a do-while loop) whose guard corresponds to the values of inputs at the initialisation of the timing chart. Indeed, the initial state of the timing chart is a stable state which does not begin with an event. The invariants to be proved for this loop correspond to the fact that outputs remain deactivated.

The events last during one scan, while stable states have an arbitrary duration and end when the next event is reached, *i.e.* when an input changes or a timer coils. In order to model this behaviour, the body of each loop iteration is enriched with an assignment of the concerned input to a random Boolean value, that may or may not update its value and lead to a new event.

```
< One scan of the Ladder program from Figure 5b >
x0 := randomb();
while (!x0 && not !x1 && not !x2 && !x3 && !x4) do
  invariant { !y70 && not !y71 && not !y72 && not !y73 && not !y74 }
  < One scan of the Ladder program from Figure 5b >
  x0 := randomb();
done
```

Fig. 6: WhyML formalisation of event ♯1 and stable state ♯(1 $\hookrightarrow$ 2)

The WhyML code of Figure 6 gives an example of formalisation of an event. It is for the event ♯1 and the stable state ♯(1 $\hookrightarrow$ 2), the latter being terminated when event ♯2 is reached, *i.e.* when `x0` is deactivated. The encoding of one scan (from Figure 5b) is intentionally duplicated. Deductive verification is unfortunately not sufficient to directly prove the invariants on outputs. Indeed, as mentioned in Section 2, the specification used to generate the formalisation lacks information on internal memory devices. To bypass this difficulty, we rely on the invariant generation plug-in for Why3 (already presented in Section 3.1) to generate additional loop invariants for each while loop of the formalisation. For instance, in each loop of the formalisation, the inference of the invariant `not !m2` would be needed.

Timer-related sequences of events. One of the most technical points of our work concerns the formalisation of fixed-duration sequences, *e.g.* events and stable states from ♯5 to ♯8 in the timing chart of the carriage line control (Figure 3).

We have to capture the fact that the total duration of this sequence is exactly 3 seconds. Since timing charts specifications do not make explicit which timer device is used to implement this aspect, we cannot, in the general case, guess which timer device appearing in the code is used for any of the fixed-duration sequences appearing in the timing chart. That is why we introduce a fresh internal counter for each fixed-duration sequence of the timing chart, add the duration constraint in the guard of each loop associated to the concerned stable states and increment the value of that counter at each loop iteration. The timer is incremented accordingly, therefore, the counter is supposed to reflect the `current` value of the timer.

In addition, there are two ways to reach the end of the loops corresponding to intermediate stable states of the fixed-duration sequences: an input change or the maximal number of scans being reached by the counter. We have to capture the fact that the termination of intermediate stable states (♯(5 $\hookrightarrow$ 6) and ♯(6 $\hookrightarrow$ 7) in our example) is due to an input update and not because the maximal number of scans has been reached. To enforce this property, we insert an `assume` clause after the loop end. In our example, we use `c1` as a counter associated with the 3 seconds fixed-duration sequence. As an illustration, we give the shape of the formalisation of event ♯5 and stable state ♯(5 $\hookrightarrow$ 6), ending with the deactivation of X4 while the number of elapsed scans of the fixed-duration sequence is not

```
< One scan of the Ladder program from Figure 5b >
x4 := randomb();
c1 := !c1 + 1;
while (not !x0 && !x1 && !x2 && not !x3 && !x4 && !c1 < 29) do
  invariant { !y70 && not !y71 && not !y72 && !y73 && not !y74 }
  < One scan of the Ladder program from Figure 5b >
  x4 := randomb();
  c1 := !c1 + 1;
done;
assume { !c1 < 29 }
```

Fig. 7: WhyML formalisation of event ♯5 and stable state ♯(5 ↪ 6)

reached yet. The resulting WhyML code is given in Figure 7. For stable state ♯(7 ↪ 8), *i.e.* the last stable state before the end of the fixed-duration sequence, there is only one way to end the loop (`!c1 >= 29`) so there is no need for any `assume` clause.

Note that the condition we use is `!c1 < 29` and not `!c1 <= 29` (or equivalently `!c1 < 30`). The reason is technical: at the end of the stable state ♯(7 ↪ 8), the counter reaches the value 29. The timer's current value is also equal to 29. The scan for event ♯8 begins and the current value of the timer is incremented during its execution (more precisely at rung 5), therefore, its value becomes equal to 30 and the timer coils.

At this stage, another pitfall remains. As explained previously, we cannot, in the general case, make explicit the equality between the introduced counter `c1` and the `current` value of the timer in the formalisation of Figure 7. Nonetheless, we can benefit from the invariant inference mechanism presented in Section 3.1. Indeed, this invariant generator does not only compute numerical domains for each variable independently: it makes use of relational domains (provided by the Apron library [13]) to infer logical relations between variables. In particular, we successfully obtain the invariant `!c1 = t0.current` that makes explicit the role of the introduced counter.

4 Implementation and Experimental Results

Our first goal is to be able to fully automatically prove that a Ladder program like our running example of in Figure 2 is conforming to a timing chart. Our secondary goal is that we want to give back, to the users, meaningful and easy-to-use information when they try to prove an incorrect implementation.

In Section 4.1, we describe the workflow of the proprietary implementation of our approach. Then, Section 4.2 presents the results obtained when executing the analysis on a correct carriage line control implementation, *i.e.*, the implementation of Figure 2. Finally, in Section 4.3, we present the feedback given by our toolbox when analysing one slight modification of the nominal program that makes the verification of conformance to the timing chart fail.

4.1 Overview of the Approach

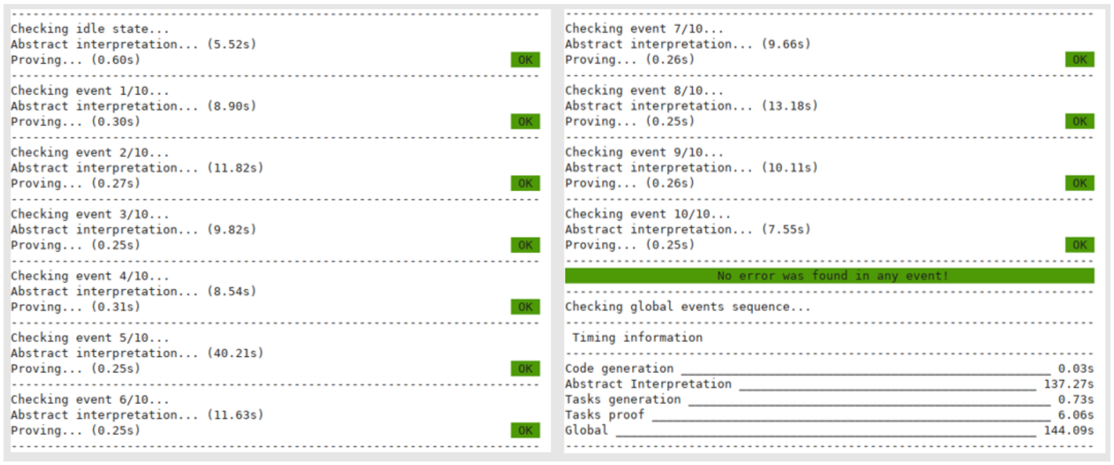
The implemented approach proceeds as follows.

1. The tool takes two inputs: an XML representation of the Ladder program, and a timing chart specification written in the PlantUML language.
2. It translates the Ladder program as a WhyML program.
3. It derives, from the timing chart the different guard conditions (hypotheses on input values) and invariants (output values to prove) for formalising the successive events of the timing chart.
4. Then, for each event,
   - Why3 infers a loop invariant for the WhyML loop that models the state that is associated to the event, thus adding information on values of internal memory to the information on output values computed in the previous step.
   - Why3 computes the verification conditions that correspond both to the inferred invariants and the invariants that correspond to the timing chart specification, and dispatches them to SMT solvers.
5. The previous step is repeated for all events. Note that besides the hypotheses on the values of inputs and outputs at the start of the event, which are given by the timing chart, the proving process also needs hypotheses on the values of internal memory values at the beginning of the event. Those values are given by the loop invariant inferred for the previous event. Hence we store, during the process, the inferred invariants for each event in order to use them as preconditions for the next event.
6. If a proof obligation fails at event n, we build a WhyML program concatenating all the previous events and the faulty one, with loops enriched with the consecutively inferred invariants. Provers are called on this WhyML program and provide counterexamples (see Section 4.3).
7. On the contrary, if all events and stable states are proved, we conclude that the Ladder program satisfies the timing chart specification.

This approach of proving each event, one by one, until a specification violation is detected, is motivated by the fact that abstract interpretation, in our examples, is far more time-consuming than proving. In the case a violation is detected, our approach avoids to launch abstract interpretation for all the events that follow the one for which the violation has been detected.

4.2 Results on correct code

We apply our approach on the nominal Ladder program described in Figure 2, for which we successfully verify the timing chart specification. Figure 8 depicts the result we obtain when running the analysis. In accordance with our strategy presented in Section 4.1, we consecutively infer invariants and then prove verification conditions for each pair (event, stable state), starting from the initial (*idle*) state of the timing chart. We observe that abstract interpretation is for now quite expensive, therefore, the proof time is negligible (6s) compared to the time for inference of invariants (137s).



```
------------------------------------------------------------------
Checking idle state...
Abstract interpretation... (5.52s)
Proving... (0.60s)                                             OK
------------------------------------------------------------------
Checking event 1/10...
Abstract interpretation... (8.90s)
Proving... (0.30s)                                             OK
------------------------------------------------------------------
Checking event 2/10...
Abstract interpretation... (11.82s)
Proving... (0.27s)                                             OK
------------------------------------------------------------------
Checking event 3/10...
Abstract interpretation... (9.82s)
Proving... (0.25s)                                             OK
------------------------------------------------------------------
Checking event 4/10...
Abstract interpretation... (8.54s)
Proving... (0.31s)                                             OK
------------------------------------------------------------------
Checking event 5/10...
Abstract interpretation... (40.21s)
Proving... (0.25s)                                             OK
------------------------------------------------------------------
Checking event 6/10...
Abstract interpretation... (11.63s)
Proving... (0.25s)                                             OK
------------------------------------------------------------------
```

```
------------------------------------------------------------------
Checking event 7/10...
Abstract interpretation... (9.66s)
Proving... (0.26s)                                             OK
------------------------------------------------------------------
Checking event 8/10...
Abstract interpretation... (13.18s)
Proving... (0.25s)                                             OK
------------------------------------------------------------------
Checking event 9/10...
Abstract interpretation... (10.11s)
Proving... (0.26s)                                             OK
------------------------------------------------------------------
Checking event 10/10...
Abstract interpretation... (7.55s)
Proving... (0.25s)                                             OK
------------------------------------------------------------------
                No error was found in any event!
------------------------------------------------------------------
Checking global events sequence...
------------------------------------------------------------------
 Timing information
------------------------------------------------------------------
Code generation ___________________________________________ 0.03s
Abstract Interpretation _________________________________ 137.27s
Tasks generation __________________________________________ 0.73s
Tasks proof _______________________________________________ 6.06s
Global __________________________________________________ 144.09s
------------------------------------------------------------------
```

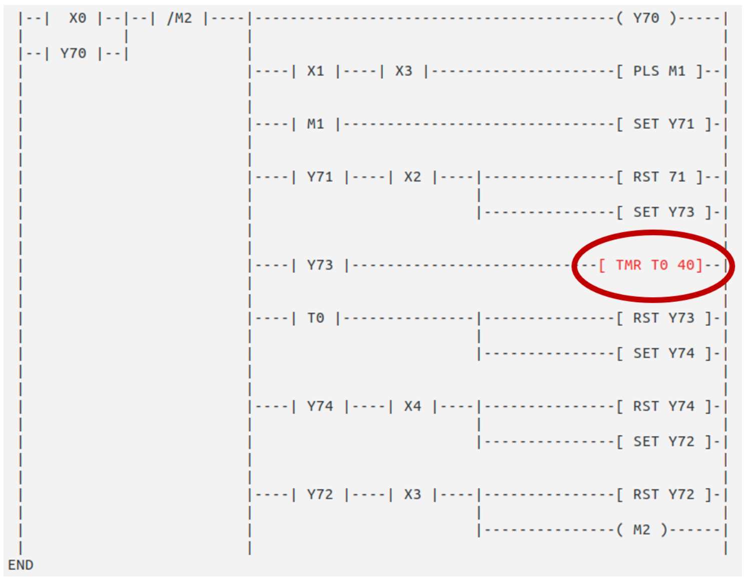
Fig. 8: Output of the tool on the nominal carriage line control program

4.3 Results on incorrect code

Let us assume the verification of a proof obligation fails for a faulty event (the case of a faulty stable state is similar). Our goal is to provide the most relevant information possible to the Ladder programmer, who may not be used to deductive verification. For that purpose, we propose an error scenario following the timing chart until the faulty event, mixing concrete values provided by counterexamples generated by Why3, and abstract domains provided by abstract interpretation.

Error scenarios. When a proof obligation fails as assumed above, Why3 is able to provide a counterexample. Since such a proof obligation comes from the verification of the concatenation of the consecutive events from the very first one to the faulty one, the information we get provides the values of the inputs, outputs and internal devices at the beginning of each event until the faulty one. Due to the way Why3 handles loops during the computation of verification conditions for SMT-solvers (that is, the loop invariant is the only known fact for the code after the loop [4]), we do not have any information on the values the devices take during the stable states. We think that this lack of information concerning values of devices during stable states may be an impediment to the understanding of the cause of the specification violation. That is why we propose to enrich the counterexample values with the domains of devices values given by abstract interpretation. This leads to the notion of *error scenario* that provides:

1. For each event that precedes the faulty one (including the faulty one), the values of devices before the beginning of the scan of this event, obtained from the counterexample trace provided by Why3.
2. For each stable state that precedes the faulty event, an over-approximation of domains of devices values, obtained by abstract interpretation.



```
|--|  X0 |--|--| /M2 |----|-----------------------------------------( Y70 )-----|
|           |             |                                                     |
|--| Y70 |--|             |                                                     |
|                         |----| X1 |----| X3 |---------------------[ PLS M1 ]--|
|                         |                                                     |
|                         |                                                     |
|                         |----| M1 |-------------------------------[ SET Y71 ]-|
|                         |                                                     |
|                         |                                                     |
|                         |----| Y71 |----| X2 |----|---------------[ RST 71 ]--|
|                         |                         |                           |
|                         |                         |---------------[ SET Y73 ]-|
|                         |                                                     |
|                         |                                                     |
|                         |----| Y73 |----------------------------[ TMR T0 40]--|
|                         |                                                     |
|                         |                                                     |
|                         |----| T0 |---------------|---------------[ RST Y73 ]-|
|                         |                         |                           |
|                         |                         |---------------[ SET Y74 ]-|
|                         |                                                     |
|                         |                                                     |
|                         |----| Y74 |----| X4 |----|---------------[ RST Y74 ]-|
|                         |                         |                           |
|                         |                         |---------------[ SET Y72 ]-|
|                         |                                                     |
|                         |                                                     |
|                         |----| Y72 |----| X3 |----|---------------[ RST Y72 ]-|
|                         |                         |                           |
|                         |                         |---------------( M2 )------|
|                         |                                                     |
END
```

Fig. 9: An incorrect version of the carriage line control

In order to convince ourselves that this notion of error scenario should be useful to Ladder programmers, we implemented different slight modifications of the carriage line control program, introducing bugs. We present one of them in this article, another one is described in the extended research report [4]. The corresponding Ladder diagram is depicted in Figure 9. The modification compared to the original code is circled.

The timer setting duration is here set to 40 scans instead of 30. We use our tool to get the reason of the proof failure, *i.e.* that Y73 is equal to true while it should be false. The obtained reason is rather intuitive: event 8 corresponds to 30 elapsed scans from timer's start. As the timer has a duration of 40 scans, it has not ended yet, therefore, Y73 is not reset yet, as highlighted by the error scenario of Figure 10a.

The trace shows that the setting value of the timer (here 40) is not reached. In particular, the current value of the timer evolves between 3 and 29 in the stable state between events 7 and 8, showing that the current event follows three other events in the fixed-duration sequence. Moreover, at the beginning of the scan of event 8, the current value of the timer and its associated counter c1 are both equal to 29, which is exactly the value we expect when leaving the fixed-duration sequence of events. The timing chart violation is depicted by Figure 10b.

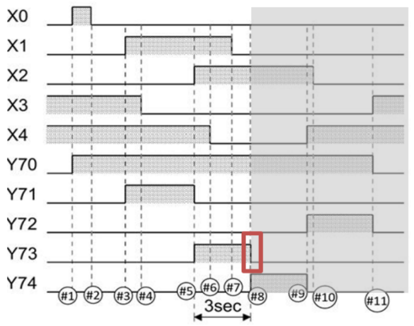
Qualitative analysis of the experiments. As a conclusion, we think that this notion of error scenario mixing concrete values provided by counterexamples to VCs, and abstract domains provided by abstract interpretation, should be useful to Ladder programmers in order to understand why a program does not conform

```
Values of devices at event 7 scan beginning:
  cc0 = false
  m1 = false
  m2 = false
  x1 = false
  y70 = true
  y71 = false
  y72 = false
  y73 = true
  y74 = false
  t0 = (current= 2, setting= 40): inactive
  c1 = 2


Values of devices between events 7 and 8:
M1 = false
M2 = false
X0 = false
X1 = false
X2 = true
X3 = false
X4 = false
Y70 = true
Y71 = false
Y72 = false
Y73 = true
Y74 = false
T0.current ∈ [3; 29]
T0.setting = 40


Values of devices at event 8 scan beginning:
  cc0 = false
  m1 = false
  m2 = false
  y70 = true
  y71 = false
  y72 = false
  y73 = true
  y74 = false
  t0 = (current= 29, setting= 40): inactive
  c1 = 29
```

(a) Output of the tool



(b) Violation of the timing chart

Fig. 10: Incorrect Ladder program: analysis results

to a given timing chart specification. A weakness of this approach is that in some cases, the concrete and abstract values might seem irrelevant. For example, for a timer counter `c`, we might have an abstract domain that states that `c` can take all the values between, say, 3 and 29 for a state, but the concrete value given for the next event might be 4. For that example, it means that the loop corresponding to the state is executed exactly once in the error scenario, before executing the next event. In that case, it might be very interesting to use the concrete values of counterexamples to refine the domain [3;29] into [3;4], and even make explicit to the programmer that there is exactly one execution of the program for the considered state.

5 Discussions, Related Work and Future Work

We presented a new method for formally verifying that a given Ladder code complies with an expected temporal behaviour expressed by a timing chart. By translating both the Ladder code and the timing chart to a WhyML program, and making use of the loop invariant generation capability of Why3, we are able to provide a fully automatic process to achieve such a verification, with a high level of confidence. Moreover, when this proof-based process fails at some point, we have a way to propose an error scenario which exposes why the Ladder code does not conform to the timing chart. Our method is implemented in a prototype which we experimented on a case study, demonstrating the effectiveness of our approach, both for formally proving the correct version and for providing counterexample scenarios on wrong mutants.

The level of confidence of our approach must be understood in terms of the trusted code base of the whole process. It first relies on the soundness of the translation from Ladder code and timing chart, which is described in Section 3. It also relies on the soundness of the VC generation process of Why3, which is not formally proven correct but validated on numerous applications [6]. Regarding trust in Why3, it is important to notice that the prototype implementation of loop invariant generation is *not* part of the trusted code base, because the loop invariants generated are later on checked for validity by the VC generation process. It is indeed fortunate to not have to rely on the soundness of this part of Why3 implementation, since we had to make significant extensions to it (mostly, support for Boolean variables, and adaptation of the API for external use) for the current purpose. The last part of the tool chain that must be trusted is the back-end SMT solver.

Regarding the generated errors scenarios, we have noticed that they are satisfactory on our case study, but due to the inherent incompleteness of counter-model generation with SMT solvers, we cannot guarantee that the generated scenarios are always valid. There are on-going work in the Why3 development team to increase the trust into the validity of generated counterexamples [3].

Related work. PLC software verification is a vast domain and numerous works have been published on that subject. The majority of them use model-checking to verify functional and temporal properties. In 2014, Ovatman *et al.* [16] published a summary of those techniques. In 2016, Darvas *et al.* [9] proposed a newer model-checking based tool and compared with former similar tools. The general drawback of the model-checking approach is that the verification it provides cannot be exhaustive, it cannot model any possible number of executions during the states of a timing chart, contrary to deductive verification. On the other hand, abstract interpretation has also been used for a long time for verifying software, in particular microcontroller software [12,15] and PLC software [5] (in combination with model-checking). Contrary to model-checking, abstract interpretation gives a full guarantee when it detects no error in a program, but it is dedicated to compute the possible values of variables during the execution of a program, and is not suited for verifying temporal properties. Finally, in a previous work [7], some of us used the Why3 deductive verification platform for detecting run-time errors of Ladder programs. This work only considered one single execution of Ladder programs and was therefore also not suited for verifying temporal properties. To our knowledge, the present paper is the first one to combine abstract interpretation and deductive verification for verifying temporal properties of Ladder programs.

Outside the context of Ladder, Stouls and Groslambert proposed an approach for proving temporal properties of C code [19], based on a translation from LTL formulas into annotations in the ACSL language [2]. These LTL formulas express temporal properties of sequences of functions calls, which are very different from our kind of specifications. Their approach is similar to ours in the sense that they automatically translate temporal properties into annotated code, to be proved correct using deductive verification. They also identified a need for automatically generating extra intermediate annotations, for which they use their own variant of abstract interpretation. A successor of this work is the CaFE plug-in of Frama-C [10], which makes use of the Frama-C plug-in EVA for abstract interpretation. Unlike us, the approaches above do not provide any facilities for explaining errors.

Future work. During our work, we had to improve the loop invariant generation feature of Why3, in particular the support for Boolean values. Even enough for our case study, there is clearly room for improvement in this implementation, required to make the tool chain more efficient. We plan to experiment our method on examples of Ladder programs that require WhyML translations involving arrays, and we have to ensure that the loop invariant generation could succeed when we are mixing all involving data-types: integers, Boolean, arrays, and also bounded integers in the future.

As mentioned at the end of Section 4.3, there is some need for improvement in the counterexample generation part of the chain. The inherent incompleteness of the SMT solvers implies that the proposed counterexample might be wrong. We are planning to incorporate in our tool-chain a recent technique that double-checks the validity of counterexamples *a posteriori* [3], which roughly amounts to symbolically executing the scenario it describes, and detect carefully at which step its behaviour diverges from what the timing chart allows.

On the error scenario side, as explained in the end of Section 4.3, the parts of a scenario that come from abstract interpretation domains, that correspond to the possible values of devices during states, could be refined using the concrete values given by the counterexamples for next events. This way, we might propose an even more understandable and useful error scenario to Ladder programmers, in case an error is detected in their code.

A longer-term goal is to augment the trust in the translation from Ladder to WhyML. We have some plans for designing a systematic and automatic validation process to confront our translation against existing test suites for Ladder programs.

References

1. Baudin, L.: Deductive verification with the help of abstract interpretation. Technical report, Univ Paris-Sud (Nov 2017), https://hal.inria.fr/hal-01634318
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.16 (2020), https://frama-c.com/html/acsl.html
3. Becker, B., Belo Lourenço, C., Marché, C.: Explaining counterexamples with giant-step assertion checking. In: Creissac Campos, J., Paskevich, A. (eds.) 6th Workshop on Formal Integrated Development Environments (F-IDE 2021). Electronic Proceedings in Theoretical Computer Science (May 2021), https://hal.inria.fr/hal-03217393
4. Belo Lourenço, C., Cousineau, D., Faissole, F., Marché, C., Mentré, D., Inoue, H.: Formal analysis of Ladder programs using deductive verification. Research Report RR-9402, Inria (Apr 2021), https://hal.inria.fr/hal-03199464
5. Biallas, S., Kowalewski, S., Stattelmann, S., Schlich, B.: Efficient handling of states in abstract interpretation of industrial programmable logic controller code. In: Proceedings of the 12th International Workshop on Discrete Event Systems. pp. 400–405. IFAC, Cachan, France (2014)
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) **17**(6), 709–727 (2015). https://doi.org/10.1007/s10009-014-0314-5, http://hal.inria.fr/hal-00967132/en, see also http://toccata.lri.fr/gallery/fm2012comp.en.html
7. Cousineau, D., Mentré, D., Inoue, H.: Automated deductive verification for ladder programming. In: Monahan, R., Prevosto, V., Proença, J. (eds.) Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019. Electronic Proceedings in Theoretical Computer Science, vol. 310, pp. 7–12 (2019). https://doi.org/10.4204/EPTCS.310.2, https://doi.org/10.4204/EPTCS.310.2
8. Dailler, S., Hauzar, D., Marché, C., Moy, Y.: Instrumenting a weakest precondition calculus for counterexample generation. Journal of Logical and Algebraic Methods in Programming **99**, 97–113 (2018). https://doi.org/10.1016/j.jlamp.2018.05.003, https://hal.inria.fr/hal-01802488
9. Darvas, D., Majzik, I., Blanco Viñuela, E.: Formal verification of safety plc based control software. In: Ábrahám, E., Huisman, M. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 9681, pp. 508–522. Springer (2016). https://doi.org/10.1007/978-3-319-33693-0_32
10. De Oliveira, S., Prévosto, V., Bardin, S.: Au temps en emporte le C. In: Baelde, D., Alglave, J. (eds.) Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015) (2015), https://hal.inria.fr/hal-01099128
11. Drath, R., Luder, A., Peschke, J., Hundt, L.: AutomationML – the glue for seamless automation engineering. In: ETFA – IEEE International Conference on Emerging Technologies and Factory Automation. pp. 616–623 (2008). https://doi.org/10.1109/ETFA.2008.4638461
12. Fehnker, A., Huuck, R., Schlich, B., Tapp, M.: Automatic bug detection in microcontroller software by static program analysis. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) Theory and Practice of Computer Science (SOFSEM). Lecture Notes in Computer Science, vol. 5404, pp. 267–278. Springer (2009). https://doi.org/10.1007/978-3-540-95891-8_26

13. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 661–667. Springer (2009)
14. Mitsubishi Electric Corporation: Mitsubishi Programmable Controllers training manual — MELSEC iQ-R Series basic course (for GX Works3). https://dl.mitsubishielectric.com/dl/fa/document/manual/school_text/sh081898eng/sh081898enga.pdf (2016), [Online; accessed 30-March-2021]
15. Nguyen, T., Aoki, T., Tomita, T., Endo, J.: Integrating static program analysis tools for verifying cautions of microcontroller. In: Asia-Pacific Software Engineering Conference (APSEC). pp. 86–93 (2019). https://doi.org/10.1109/APSEC48747.2019.00021
16. Ovatman, T., Aral, A., Polat, D., Ünver, A.: An overview of model checking practices on verification of PLC software. Software & Systems Modeling **15**, 1–24 (12 2014). https://doi.org/10.1007/s10270-014-0448-7
17. Ramanathan, R.: The IEC 61131-3 programming languages features for industrial control systems. In: World Automation Congress (WAC). pp. 598–603 (2014). https://doi.org/10.1109/WAC.2014.6936062
18. Roques, A.: PlantUML standard library. https://plantuml.com/stdlib (2009), [Online; accessed 24-March-2021]
19. Stouls, N., Groslambert, J.: Vérification de propriétés LTL sur des programmes C par génération d'annotations. Research report (2011), https://hal.inria.fr/inria-00568947