Automated Formal Analysis of Temporal Properties of Ladder Programs

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

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automated Formal Analysis of Temporal Properties of Ladder Programs

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

the date of receipt and acceptance should be inserted later

Abstract Programmable Logic Controllers are industrial digital computers used as automation controllers in manufacturing processes. The Ladder language is a programming language used to develop software for such controllers. In this work, we consider the description of the expected behaviour of a Ladder program under the form of a *timing chart*, describing a scenario of execution. Our aim is to prove that the given Ladder program conforms to the expected temporal behaviour given by such a timing chart. Our approach amounts to translating the Ladder code, together with the timing chart, into a program for the Why3 environment for deductive program verification. The verification proceeds with the generation of *verification conditions*: mathematical formulas to be checked valid using automated theorem provers. The ultimate goal is two-fold. On the one hand, by obtaining a complete proof, one verifies the conformity of the Ladder code with respect to the timing chart with a high degree of confidence. On the other hand, in the case the proof is not fully completed, one obtains 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

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

Cláudio Belo Lourenço
Université Paris-Saclay, CNRS, Inria, LMF, 91190, Gif-sur-Yvette, France

Denis Cousineau
Mitsubishi Electric R&D Centre Europe, Rennes, France

Florian Faissole
Mitsubishi Electric R&D Centre Europe, Rennes, France

Claude Marché
Université Paris-Saclay, CNRS, Inria, LMF, 91190, Gif-sur-Yvette, France

David Mentré
Mitsubishi Electric R&D Centre Europe, Rennes, France

Hiroaki Inoue
Mitsubishi Electric Corporation, Amagasaki, Japan

1 Introduction

Programmable Logic Controllers (PLCs for short) 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 [29] among other languages [9] for programming PLCs. The Ladder language is still widely used and very popular among technicians and electrical engineers.

Because of the widespread use 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 a scenario of execution. Timing charts are commonly used in the industry to specify small to medium-sized programs, like in the *Function Blocks* libraries that are shipped by PLC manufacturers together with their programming software. Our approach consists of automatically translating the Ladder code, and the timing chart all together, into a program written in the WhyML language, which is the input language of the generic Why3 environment for deductive program verification [8]. In WhyML, expected behaviours of program are expressed using *contracts*, which are annotations expressed in formal logic. The Why3 environment provides a set of software tools for checking that the WhyML code conforms to these formal contracts. This verification process is performed using automated theorem provers as back-ends, so that at the end, if the back-end proof process succeeds, the conformity 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 goal being to build a tool that is useful to regular Ladder programmers. More precisely, in the 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 article is an extension of a former version published in the proceedings of the FMICS conference (*Formal Methods for Industrial Critical Systems*) in 2021 [5]. The contributions and the structure of this article are as follows. Section 2 is a preliminary section introducing the basics of Ladder programming, and the way their expected temporal behaviours are expressed using timing charts. It is illustrated on a running example which will also serve as experimental evaluation. It contains a few extra technical explanations w.r.t. the FMICS paper. Section 3 is also partly a preliminary section, introducing the main Why3 features that are needed for the translation of Ladder code, and timing charts, into WhyML programs. This section is significantly extended w.r.t. the FMICS paper, so as to give details on two aspects: the loop invariant generation and the counterexample generation. The counterexample generation is a pre-existing feature of Why3 that we use as is, whereas the loop invariant generation (detailed in Section 3.1) is a contribution of ours, built upon a pre-existing prototype, that we extended and made more robust, in particular regarding the support for Boolean variables which

appeared to be of utmost importance for our work. We then present in Section 4 our second contribution, namely a translation scheme of Ladder code and timing charts into WhyML. This section is augmented (w.r.t. the FMICS paper) with all the needed auxiliary Why3 functions to represent Ladder nodes, in particular timers, and with a more generic and detailed presentation of the translation of timing charts specifications. A third contribution is our implementation and our experimentation, presented in Section 5. The experimental results are described in two different sides: first a case of a complete proof success and second two variants exhibiting a proof failure, where in both cases some counterexample scenario is generated. The FMICS paper was presenting only one case of failure, we present here another one, differing from the first in the sense that it highlights the interest of using *abstract interpretation* domains, besides counterexample values, for building error scenarios. In addition to the FMICS paper, Section 5.5 also presents a graphical user interface we built, which allows the user to browse error scenarios and animate the corresponding erroneous executions. A final new contribution of this article is a discussion of industrial applicability of our method, in Section 5.6. We discuss related work and future work in Section 6.

2 Quick Introduction to Ladder Programming

A Ladder program is called a *diagram*, as it is usually presented in a graphical manner. A rather simple example of a PLC controlling a Carriage line is depicted in Figure 1, with the corresponding Ladder diagram in Figure 2. We will use this as a running example in this paper. We detail the example itself in the next subsection.

Generally speaking, a Ladder diagram manipulates three different kinds of memory locations (called *devices*), namely *inputs* (which names start with an X) that correspond to the input wires of the PLC, *internal devices* (which names start with an M, a D, etc...) that correspond to the internal memory of the PLC, and *outputs* (which names start with a Y) that correspond to the output wires of the PLC. Inputs and outputs are given Boolean values, while internal devices can furthermore contain integer values, floating-point values, string values, *etc.* *Contacts* are used to read the value of a device. *Coils* are used to write a value to a device. Besides coils, *instructions* can also be used to write values to devices, given the values of other devices.

Graphically, contacts are depicted by the notation `-| |-` and are located at the left of the diagram, while coils are depicted by the notation `-< >-` (or `-( )-`) and are located at the right of the diagram together with instructions calls which are depicted by the notation `-[ ]-`. Negation is depicted by the notation `/` and can be used directly in a contact or before a coil or an instruction call. Contacts can be combined in a serial way (Boolean conjunction) or in a parallel way (disjunction). Coils write, to their associated device, the value corresponding to the combinations of contacts at their left. We call this combination of contacts the *front* of the coil. Instructions are activated when their front gives the *true* value. Coils and instructions can also be parallelised. A line with contacts, coils and instructions is called a *rung*, and a program (a *diagram)* might be composed of several rungs. Such a Ladder program is executed cyclically in a synchronous way: first, values of the input wires of the PLC are written to the corresponding input devices, then the

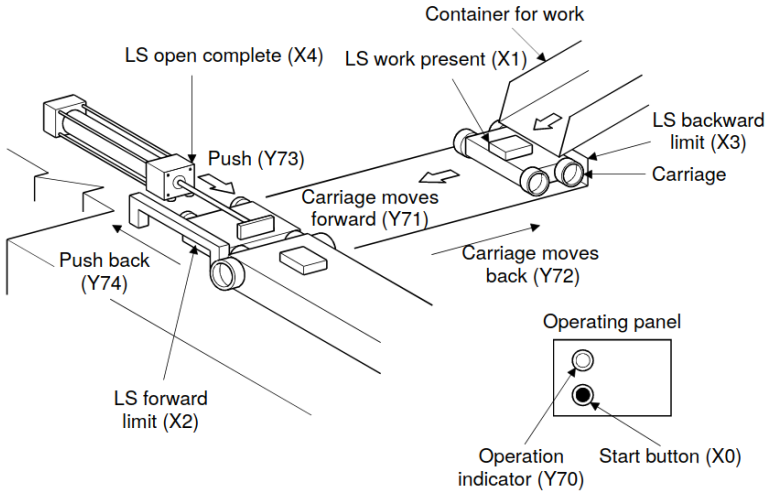


Fig. 1: Carriage line control: System description.

program is executed and eventually output wires are activated according to the values of the output devices after execution. One single execution of the program is called a *scan*.

Besides regular memory devices, there are also special internal devices, *e.g.*, *timer* devices. The threshold of a timer is expressed in terms of a number of scans (meaning that its actual duration depends of the frequency of the execution of the Ladder program in the PLC). When manipulating a timer device, a coil is used to set the threshold and increment the internal counter, and a contact is used to determine whether the counter has reached the threshold or not. Note that, contrary to instructions, timer coils have an impact on memory when their front is not activated: in that case, the associated counter is reset to 0. Hence, a timer coil must be consecutively activated during a number of scans greater than the associated threshold for the corresponding timer contact to be activated.

2.1 Description of the Carriage Line Control Example

This example comes from a Mitsubishi Electric training manual for programming PLCs [26]. We illustrate some principles of Ladder Logic on the first rung of this example: this rung expresses the fact that the output Y70 receives the value of the Boolean formula

$$(\mathrm{X0} \vee \mathrm{Y70}) \wedge (\neg\, \mathrm{M2})$$

i.e., if the corresponding physical devices are activated such that the Boolean formula is true, then Y70 is activated, and it 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

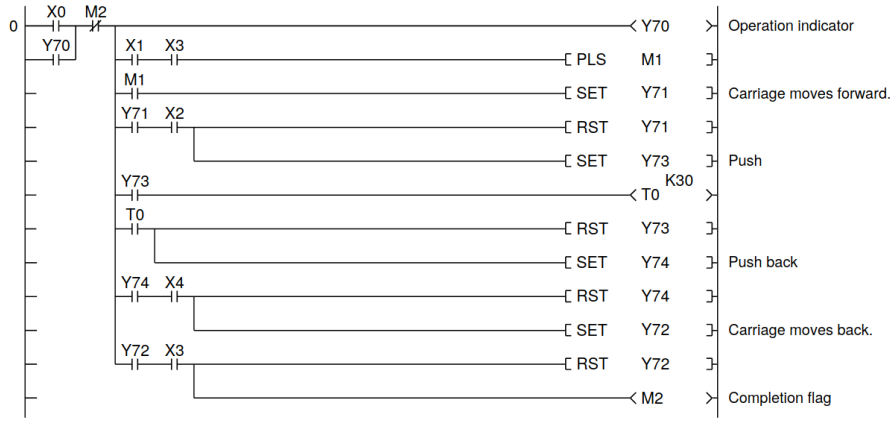


Fig. 2: Carriage line control: Ladder program.

device or an output device) when its front 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 (reset) instruction is the opposite: the device argument is deactivated when the front is activated. The PLS (pulse) instruction activates its device argument on a rising edge of its front, *i.e.*, when its front is activated for the current scan while it was not for the previous one.

The diagram also uses a timer device T0. The timer coil designates 30 as its threshold (notation K30 means "constant value 30"). After 30 consecutive scans in which the timer coil for T0 is activated, the timer contact for T0 is activated and remains activated until the front of the timer coil is deactivated. Each time the front of the timer coil is deactivated, the counter of the timer is reset to 0.

2.2 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 [16] 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. To our knowledge, timing charts are generally used to specify programs of comparable size as our example, *e.g.*, *Function Blocks*, which are kind of library functions that are shipped together with a PLC and a programming environment. See Section 5.6 for a detailed discussion on the use of timing charts in industrial applications. A timing chart specifies the evolution of the PLC outputs over the execution of scans, according to the evolution of the PLC 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 *states*, *i.e.*, arbitrary-length successions of scans in which the values of both inputs and outputs are unchanged.

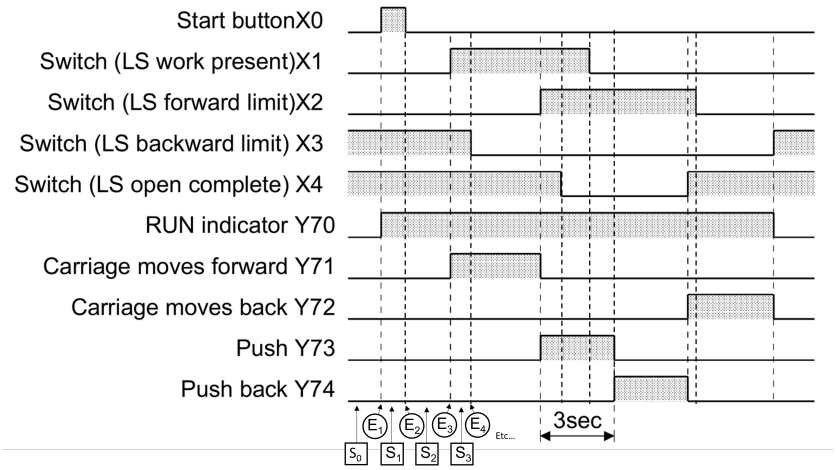


Fig. 3: Timing chart specification for the Carriage line control.

Figure 3 depicts the considered timing chart specification of the Carriage line control example. Events of the timing chart are depicted as $E_1, E_2, E_3, E_4, \ldots$ while the associated states are depicted as S_0, S_1, S_2, S_3, etc. For each event E_i, S_i denotes its following state, while S_0 is the initial state (the *idle state* of the system). The timing chart of Figure 3 also contains a fixed-duration sequence of events and states, whose duration is 3 seconds, represented by the arrow $\overset{3\text{sec}}{\longleftrightarrow}$ between events E_5 and E_8. We call *fixed-duration sequence* the corresponding sequence of events and states. Typically, the Ladder program is executed periodically every 100 milliseconds, therefore, the fixed duration 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-level verification [11]. 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*.

```
1   val b : ref bool
2   val x : ref int
3
4   let toy () : unit
5     requires { 0 <= !x <= 10 }
6     writes { b, x }
7     ensures { not !b }
8     ensures { !x <= 200 }
9   =
10    b := false;
11    while (!x < 100) do
12      b := (!x < 50);
13      if !b then x := !x + 2
14            else x := !x + 3;
15    done;
16    assert { !x >= 75 }
```

Fig. 4: Toy example of a WhyML program with a formal contract.

3 The Why3 Environment

Why3 is a generic environment for deductive program verification, providing the language WhyML for specification and programming [19,8,20]. The genericity of Why3 is exemplified by the fact that WhyML is already used as an intermediate language for verification of programs written in C, Java, Ada or Rust [18,22]. The specification component of WhyML [7], used to write program annotations and background theories, is an extension of first-order logic. The specification part of the language serves as a common format for theorem proving problems, *proof tasks* in Why3's jargon. Why3 generates proof tasks from user lemmas and annotated programs, using a weakest-precondition calculus, then dispatches them to multiple provers. It is indeed another aspect of the genericity of Why3: its ability to dispatch proof tasks to many different provers. In practice, for the proof of programs, provers of the SMT (*Satisfiability Modulo Theories*) family are the most successful ones, indeed at least if they support quantified formulas, which is the case for Alt-Ergo [10], CVC4 [1] and Z3 [14].

We illustrate the essential Why3 features on the toy WhyML program presented in Figure 4. This code involves two global variables, `b` of type Boolean and `x` of type integer, that is a mathematical, unbounded integer in WhyML. The function `toy` takes no arguments, and is equipped with a *formal contract*. This contract first involves a *precondition*, introduced by keyword `requires` on line 5, stating that the value of `x` on function entry is required to lie between 0 and 10. This contract also involves two postconditions, introduced by keyword `ensures` on lines 7-8, stating respectively that at exit, `b` is false and `x` is smaller than or equal to 200. The clause `writes` on line 6 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 on line 16, 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 valid, only the one for the assertion is proved: indeed, this assertion directly follows from the negation of the loop condition, which entails that the value of `x` is not smaller than 100. 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*. For example, we can manually add such an invariant in our toy example as follows:

```
...
while (!x < 100) do
  invariant { 0 <= !x <= 150 }
  b := (!x < 50);
  ...
```

With that change, Why3 now generates additional VCs, stating that this invariant initially holds when entering the loop, and that it is preserved by an arbitrary iteration of the loop body. These extra VCs are proved valid, and now the VC corresponding to the second post-condition is proved valid too, because it is a consequence of the loop invariant being true at exit of the loop. Yet, the first post-condition is still not proved valid, there is a need to make the loop invariant stronger. This could be done by hand, but indeed we show below that it can be done by our procedure for automatic generation of loop invariants.

There are indeed two recently added features of Why3 that are of particular interest for our work on Ladder programs. The first one is the capability of *automatically generating loop invariants*. This was initially an experimental prototype feature [2], that was extended and made more robust by us, precisely for the purpose of this work. The second one is the capability of *generating counterexamples* from failed proof attempts. This second feature is originally motivated by other industrial applications, in particular the use of Why3 inside the Spark/Ada environment [23] for the development of safety-critical applications. We detail these two features below.

3.1 Automatic Generation of Loop Invariants

The WhyML code that will be automatically generated from Ladder code and an associated timing chart is mostly a sequence of while loops, operating on quite a large number of variables. As seen above, and as usual in deduction-based program verification, proving those programs with loops requires to insert appropriate loop invariants, that in principle should be written by the user.

Nevertheless, in the context of this paper, the objective is to build a fully automated tool for bringing deductive verification of timing charts specifications to Ladder developers, for whom writing the appropriate loop invariants by hand would be a tedious and difficult task. Fortunately, there are existing methods to automatically infer such invariants, in particular the technique of *abstract interpretation*, which roughly consists in computing an over-approximation of all the reachable program states, from any possible execution. For our work, we built upon a former prototype for Why3 designed in 2017 [2]. This prototype uses the

Apron [21] library to support integer abstraction domains such as intervals, octagons or polyhedra.

In fact, it appeared that the prototype above was not sufficiently complete for our purpose. Apart from various technical improvements, we had to add some dedicated support for Boolean variables. Boolean variables indeed appear everywhere in the translation of Ladder programs, and the former prototype was not generating any invariants for them, making the invariants generated insufficient at first. The extended implementation that we realized is now part of the main branch of development of Why3, and already released in Why3 version 1.5. This new support for Boolean variables works by encoding them as integers variables with appropriate constraints, so that the underlying chosen numerical domain is able to take care of them. It enables in particular, for relational domains like octagons or polyhedra, the inference of relations between Boolean variables and integer variables. As we discuss in Section 6.2, this approach is not optimal and could be made more efficient in the future. Apart from this special handling of Boolean values, the invariant generation proceeds in a standard way from the abstract interpretation theory: a control-flow graph is computed, and the abstract states reachable in each graph node are computed thanks to a fix-point computation.

We illustrate the invariant generation process on our toy WhyML program above. Instead of adding a loop invariant by hand as we did before, we instruct Why3 to automatically infer such a loop invariant. In practice we have to declare the domain we want to use, here we use the polyhedra domain. The generated loop invariant is then as follows:

```
...
while (!x < 100) do
  invariant { (!b = false ∧ 0 <= !x <= 10) ∨
              (!b = true  ∧ 2 <= !x <= 51) ∨
              (!b = false ∧ 53 <= !x <= 102)
  }
  b := (!x < 50);
  ...
```

Notice in particular how this invariant relates the values of the Boolean `b` and the integer `x`. With this loop invariant added, all the VCs generated are automatically proved valid, including the initialisation and preservation of that loop invariant, and also the first post-condition that was not proved with the manually added loop invariant.

An important final remark on the invariant generation is about the guarantee of its soundness. This is an important question since we rely on an implementation which remains in a quite prototype state, and makes use of ad-hoc extensions to support Boolean values. Fortunately, in the current Why3 verification process, the generated invariants are *not trusted*, but double-checked by the core VC generation process, that is, a generated invariant is checked valid at loop entry, and preserved by an arbitrary loop iteration. In other words, our prototype of automatic generation of loop invariants does not risk to compromise the soundness of the formal verification.

3.2 Generation of Counterexamples

When a given proof task fails to be proved, we need to have some feedback so as to understand why it is not proved. In Why3, when a proof is attempted with an SMT solver, and the goal is not proved, then Why3 implements an approach that can propose a *potential counterexample* to that proof attempt, expressed at the level of the WhyML source code [12]. Such a counterexample is only said potential since it may be spurious, because of the incompleteness of SMT solvers and also the incompleteness of the VC-based verification approach. There are recent experimental methods to check the validity of counterexamples *a posteriori* [4] but these are not yet integrated in our tool-chain.

Let's illustrate this feature again on our toy WhyML program. Assume that we introduce a modification of the code, changing the loop condition as follows.

```
...
while (!x < 300) do
  ...
```

Still assuming that we ask to generate 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 (in fact Why3 also provides values taken by variables at previous steps of execution [12, 4]). These values for `b` and `x` indeed satisfy the loop invariant, but with those the post-condition `!x <= 200` is not valid.

The overall process for counterexample generation in Why3 is described by Dailler *et al.* [12]. Roughly speaking, it consists in getting a counter-model from the SMT solver and inverting the VC generation process to obtain a counterexample at the level of the WhyML source code. This does not suffice for obtaining a counterexample at the level of Ladder programs: to use the counterexample feature in our use case of verifying Ladder code, we had to develop an additional procedure to turn the potential counterexample proposed by Why3 into a meaningful counterexample at the level at the Ladder code and the timing chart. Although this process is not very complex, it has been mandatory to develop a few extensions to Why3, in particular regarding the traceability. Thanks to these extensions, Why3 now proposes to every user of WhyML as an intermediate language a better mechanism to relate an input language to its WhyML encoding, and make use of this to more easily reconstruct a counterexample at the input source level. The presentation of the counterexamples at the level of Ladder will be described in Section 5.3 below.

4 Translation of Ladder Programs to WhyML

Concretely, our tool automatically translates Ladder programs given as XML files and timing charts given as PlantUML [30] files into WhyML programs. We describe in Section 4.1 how we translate the Ladder program itself, and in Section 4.2 how we use this translation for modeling the successive executions of the program and verifying that it satisfies the given timing chart.

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

Fig. 5: WhyML function for SET instruction.

```
let rst (front : bool) (device : ref bool) : unit
  writes  { device }
  ensures { !device ↔ (not front ∧ (old !device)) }
  =
  if front then device := false
```

Fig. 6: WhyML function for RST instruction.

```
let pls (front : bool) (m : ref bool) (cc_front : ref bool) : unit
  writes { m, cc_front }
  ensures { !m = (front && (not !(old cc_front))) ∧ !cc_front = front }
  =
  m := (front && not !cc_front) ;
  cc_front := front
```

Fig. 7: WhyML function for PLS instruction.

4.1 Translation of the Ladder Program

Our translation uses the Why3 Boolean library to model front combinations and relies on predefined models of Ladder instructions as WhyML functions. Figure 5 depicts the function that corresponds to the SET instruction. This function takes two arguments, first the front value 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 function for the RST (reset) instruction, presented in Figure 6, is very similar.

The WhyML function for the instruction PLS (pulse) is given in Figure 7. Besides the front and device arguments, it introduces another variable argument (`cc_front`) that denotes the value of the front at the previous scan. Then it uses this value to activate the device argument if and only if the front argument is activated at this scan while it was not at the previous scan. As one can see, memorisation of values at previous scan of devices and fronts is done only for the needed ones (here, directly by the called instruction PLS).

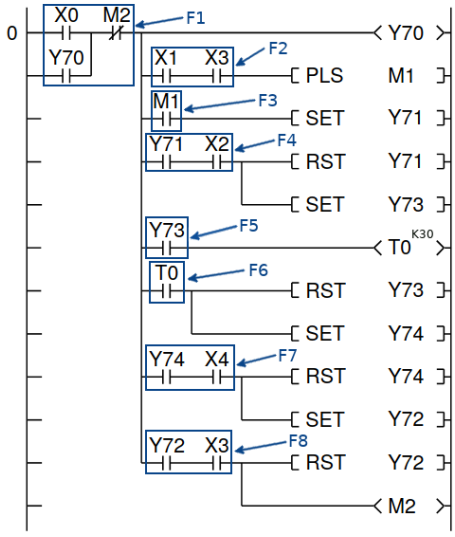
Figure 8 describes how we model timers. First we use a record type with two fields: `current` that gives the current value of the timer counter and `setting` that gives the threshold of the timer. Then, we model the timer coil with the `timer_coil` function. As for Ladder timers, this function, when activated, sets the threshold

```
type timer = {
  mutable current : int;
  mutable setting : int;
}

let timer_coil (front : bool) (t : timer) (v : int) : unit
  writes { t }
  ensures {  t.setting = v
             ∧ ((front ∧ t.current = (old t.current) + 1)
             ∨ ((not front) ∧ t.current = 0)) }
  =
  if front then (t.current ← (t.current + 1); t.setting ← v)
           else (t.current ← 0; t.setting ← v)

let timer_contact (t : timer) : bool
  ensures { result = (t.current >= t.setting) }
  =
  (t.current >= t.setting)
```

Fig. 8: WhyML code for timers.



(a) Ladder code with common fronts.

```
let scan () =
  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
```

(b) WhyML encoding.

Fig. 9: Encoding of one scan of the Ladder program for the Carriage line control.

of the timer and increment its counter. When its front is deactivated, it resets the counter to zero (remind that the timer coil should be activated during a number of consecutive scans that correspond to the threshold, for the timer to end). The `timer_contact` function models the fact that the timer device is activated when its counter reaches its threshold value.

Given this WhyML modelling of Ladder instructions, we can now give, in Figure 9b, 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 in Figure 9a.

4.2 The Ladder Loop, and the Encoding of Timing Charts

By definition, timing charts are made of successive events and 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 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 automated process that takes a Ladder diagram and a timing chart specification and returns the corresponding WhyML modelling.

4.2.1 Events and states as loops

Let us describe the WhyML modelling of an event E_i and its associated state S_i. We suppose here that the event is not part of a timer-related sequence, this case is described in Section 4.2.2. We write X_i the input whose value changes at event E_i (*e.g.*, X_1 is X0). We write G_i the formula being the conjunction of value assertions concerning inputs for event E_i and state S_i (*e.g.*, G_1 is X0∧¬X1∧¬X2∧X3∧X4). We write I_i the formula being the conjunction of value assertions concerning outputs for event E_i and state S_i (*e.g.*, I_1 is Y70∧¬Y71∧¬Y72∧¬Y73∧¬Y74).

The WhyML modelling of a pair of an event E_i and a state S_i uses a *do-while* style loop[1] to model the fact that events correspond to exactly one execution while states can last an arbitrary number of executions (possibly zero). The body of the loop corresponds to the WhyML modelling of one scan of the Ladder program. The guard of the loop corresponds to the assumptions on inputs, *i.e.*, the formula G_i regarding the values taken by the inputs at E_i and S_i. The verification conditions on outputs are modelled as a loop invariant I_i: the invariant initialisation corresponds to the event while its preservation corresponds to the state. To model the possibility of reaching the next event after (at least) one execution (corresponding to the event), we enrich the WhyML modelling of the program with an assignment of the input that changes at next event (X_{i+1}) to a random Boolean value, that may or may not update its value and lead to event E_{i+1}. In other words, if we write `scan()` the WhyML modelling of one scan of the program, and we assume that we have a function named `random` that gives a random Boolean value to its argument, the modelling of event E_i and state S_i is given in Figure 10. The WhyML code of Figure 11 gives the more concrete modelling of event E_1 and state S_1 for our running example.

The idle state S_0 is handled almost the same way, except that it uses (only) a while loop, since its duration could be zero. The guard condition is given by the input values during S_0 (in our running example, G_0 is ¬X0∧¬X1∧¬X2∧X3∧X4). The invariant is given by the output values during S_0 (in our running example

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

```
(* event E_i *)
scan();
random(X_{i+1});

(* state S_i *)
while G_i do
  invariant { I_i }
  scan();
  random(X_{i+1});
done
```

Fig. 10: Pseudo-code of the WhyML modelling of event E_i and state S_i.

```
scan(); (* Figure 9b *)
x0 := randomb();
while (!x0 && not !x1 && not !x2 && !x3 && !x4) do
  invariant { !y70 && not !y71 && not !y72 && not !y73 && not !y74 }
  scan(); (* Figure 9b *)
  x0 := randomb();
done
```

Fig. 11: WhyML modelling of event E_1 and state S_1 for the Carriage line control.

```
while (not !x0 && not !x1 && not !x2 && !x3 && !x4) do
  invariant { not !y70 && not !y71 && not !y72 && not !y73 && not !y74 }
  scan(); (* Figure 9b *)
  x0 := randomb();
done
```

Fig. 12: WhyML modelling of state S_0 for the Carriage line control.

all outputs should be deactivated before the execution of the program hence I_0 is $\neg$Y0$\wedge\neg$Y1$\wedge\neg$Y2$\wedge\neg$Y3$\wedge\neg$Y4). We depict in Figure 12 the WhyML modelling of state S_0 for our running example.

Finally, in the absence of timer-related sequence of events, the full WhyML modelling that models the execution of the program according to the timing chart is given by the sequence of the WhyML modellings of S_0, then E_1 and S_1, then E_2 and S_2, *etc.*

As mentioned in Section 2.2, the specification used to generate the WhyML modelling lacks information on internal memory devices to be able to automatically prove the presented invariants. To bypass this difficulty, we rely on the invariant generation plug-in for Why3 (presented in Section 3.1) to generate additional loop invariants for each while loop of the modelling. For instance, in each loop of the WhyML code, the inference of the invariant (`not !m2`) is needed and indeed provided by Why3's invariant generation plug-in.

4.2.2 Timer-related sequences of events

One of the most technical points of our modelling concerns fixed-duration sequences, *e.g.*, events and states from E_5 to E_8 in the timing chart of the Carriage line control (Figure 3).

Let us write $T_{i\to j}$ such a sequence between events E_i and E_j. The events and states that are contained in $T_{i\to j}$ are all the E_k and S_k with $i \leq k < j$ and E_j. In our example, $T_{5\to 8}$ contains E_5, S_5, E_6, S_6, E_7, S_7 and E_8. Such a sequence is associated to a timing that can be converted to a number of executions, given the frequency of executions applied by the PLC. Let us write $n_{i\to j}$ the number of executions for sequence $T_{i\to j}$. In our example, in the typical case, $n_{5\to 8} = 30$.

Our goal is to capture the fact that the total number of executions during $T_{i\to j}$ is exactly $n_{i\to j}$. In other words, we have to capture that the sum of numbers of executions during S_i,...S_{j-1} is equal to $n_{i\to j} - (j + 1 - i)$ (the $(j + 1 - i)$ lasting executions corresponding to events E_i,...,E_j). In our example, $n_{5\to 8} = 30$ and we want the sum of numbers of executions during S_5, S_6 and S_7 being equal to 26.

Since timing charts specifications do not make explicit which timer device should be 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 and increment its value after each execution corresponding to E_k or contained in the loop corresponding to S_k with $i \leq k < j$. This way, the counter value reflects the `current` value of the timer that is used to implement the sequence. Let us write $c_{i\to j}$ the counter associated to $T_{i\to j}$. In the modelling of states S_i to S_{j-1}, we have to enrich the guard condition to specify that the timer has not reached its threshold too early, since the threshold is supposed to be reached exactly at event E_j. We therefore enrich the guard condition with the formula $c_{i\to j} < n_{i\to j} - 1$. Additionally, to specify that the loops for states S_i to S_{j-2} should not exit because of that last condition, we add an `assume` clause stating that $c_{i\to j} < n_{i\to j} - 1$ after those loops. There should not be any random assignment in the end of the body of state S_{j-1} since X_j is not defined (event E_j is not triggered by an input change but by the ending of the fixed duration). This means that the only way to exit the loop of S_{j-1} is to contradict $c_{i\to j} < n_{i\to j} - 1$ (and we get $c_{i\to j} = n_{i\to j} - 1$).

The pseudo-code of the modelling of a fixed-duration sequence is depicted in Figure 13 and the code of the modelling of the fixed-duration sequence $T_{5\to 8}$ of our running example is given in Figure 14. Nnote that the counter $c_{5\to 8}$ is here written `c1`.

Note that our modelling adapts well to timing charts that contains several fixed-duration sequences (if those sequences do not overlap) thanks to the fact that we do not have to guess which timer device is used for which fixed-duration sequence. Now, for formally proving that the program satisfies the timing chart specification, a last pitfall remains: link the introduced counter with the associated timer `current` field in the Why3 modelling, and be able to prove that they always have the same value. Fortunately, 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 to infer logical relations between variables. In particular,

```
(* event E_i *)
scan();
random(X_{i+1});
c_{i→j}++;

(* state S_i *)
while G_i && c_{i→j} < n_{i→j} − 1 do
  invariant { I_i }
  scan();
  random(X_{i+1});
  c_{i→j}++;
done
assume { c_{i→j} < n_{i→j} − 1 }

(* event E_{i+1} *)
scan();
random(X_{i+2});
c_{i→j}++;

(* state S_{i+1} *)
while G_{i+1} && c_{i→j} < n_{i→j} − 1 do
  invariant { I_{i+1} }
  scan();
  random(X_{i+2});
  c_{i→j}++;
done
assume { c_{i→j} < n_{i→j} − 1 }

...

(* event E_{j−1} *)
scan();
c_{i→j}++;

(* state S_{j−1} *)
while G_{j−1} && c_{i→j} < n_{i→j} − 1 do
  invariant { I_{j−1} }
  scan();
  c_{i→j}++;
done

(* event E_j *)
scan();
random(X_{j+1});
```

Fig. 13: Pseudo-code of the modelling of sequence $T_{i \to j}$.

for each loop contained in a fixed-duration sequence, we successfully obtain the invariant `!c1 = t0.current` that makes explicit the role of the introduced counter.

5 Implementation and Experimental Results

We remind that our first goal is to be able to automatically prove that a Ladder program like our running example of Figure 2 is conforming to its associated timing

```
(* event E5 *)
scan();
x4 := randomb();
c1 := !c1 + 1;

(* state S5 *)
while (not !x0 && !x1 && !x2 && not !x3 && !x4 && !c1 < 29) do
  invariant { !y70 && not !y71 && not !y72 && !y73 && not !y74 }
  scan();
  x4 := randomb();
  c1 := !c1 + 1;
done;
assume { !c1 < 29 }

(* event E6 *)
scan();
x1 := randomb();
c1 := !c1 + 1;

(* state S6 *)
while (not !x0 && !x1 && !x2 && not !x3 && not !x4 && !c1 < 29) do
  invariant { !y70 && not !y71 && not !y72 && !y73 && not !y74 }
  scan();
  x1 := randomb();
  c1 := !c1 + 1;
done;
assume { !c1 < 29 }

(* event E7 *)
scan();
c1 := !c1 + 1;

(* state S7 *)
while (not !x0 && not !x1 && !x2 && not !x3 && not !x4 && !c1 < 29) do
  invariant { !y70 && not !y71 && not !y72 && !y73 && not !y74 }
  scan();
  c1 := !c1 + 1;
done;

(* event E8 *)
scan();
x4 := randomb();
```

Fig. 14: WhyML modelling of fixed-duration sequence $T_{5\to 8}$.

chart given in Figure 3. 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.

Section 5.1 describes the workflow of the proprietary implementation of our approach. Section 5.2 presents the results obtained when executing the analysis on a correct Carriage line control implementation, *i.e.*, the implementation of Figure 2. Section 5.3 presents the feedback given by our toolbox when analysing slight modifications of the nominal program that makes the verification of conformance to the timing chart fail. Section 5.4 discusses the former results. Section 5.5 details the graphical interface we designed to help the user browse into error scenarios.

Section 5.6 concludes by a general discussion on the industrial applicability of our approach.

5.1 Overview of the Approach

The implemented approach proceeds as follows. Note that for performances reasons, our implementation differs slightly from what we described in Section 4: instead of translating the whole timing chart at once, we process each pair of an event and a state, one after one.

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 into WhyML as described in Section 4.1.
3. It derives, from the timing chart, the different guard conditions (hypotheses on input values) and invariants (output values to prove), as described in Section 4.2 for modelling in WhyML the successive events of the timing chart.
4. Then, for each pair of an event and its following state,
   - 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 state. Hence we store, during the process, the inferred invariants for each state 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 hopefully provide a counterexample (see Section 5.3).
7. On the contrary, if all events and states are proved, we conclude that the Ladder program satisfies the timing chart specification.

This approach of proving each event and state, one by one, until a specification violation is detected, was motivated by the fact that first versions of Why3's abstract interpretation plugin were time-consuming. In the case a violation was detected, our approach avoided to launch abstract interpretation for all the events that follow the one for which the violation was detected.

As seen in Figure 15, Why3's abstract interpretation plug-in has improved a lot since our first experiments [5]: timings have been divided by almost 8 on our example. Those better performances could allow us to use a simpler approach that would directly try to infer invariants and launch proving on the whole WhyML program modelling all events and states at once.

```
------------------------------------------------------------------------
Checking idle state...
Abstract interpretation... (0.56s)
Proving... (0.43s)                                                   OK
------------------------------------------------------------------------
Checking event 1/10...
Abstract interpretation... (1.02s)
Proving... (0.42s)                                                   OK
------------------------------------------------------------------------
Checking event 2/10...
Abstract interpretation... (0.95s)
Proving... (0.42s)                                                   OK
------------------------------------------------------------------------
Checking event 3/10...
Abstract interpretation... (2.05s)
Proving... (0.42s)                                                   OK
------------------------------------------------------------------------
Checking event 4/10...
Abstract interpretation... (0.97s)
Proving... (0.42s)                                                   OK
------------------------------------------------------------------------
Checking event 5/10...
Abstract interpretation... (5.45s)
Proving... (0.45s)                                                   OK
------------------------------------------------------------------------
Checking event 6/10...
Abstract interpretation... (1.49s)
Proving... (0.45s)                                                   OK
------------------------------------------------------------------------
Checking event 7/10...
Abstract interpretation... (1.21s)
Proving... (0.45s)                                                   OK
------------------------------------------------------------------------
Checking event 8/10...
Abstract interpretation... (1.44s)
Proving... (0.45s)                                                   OK
------------------------------------------------------------------------
Checking event 9/10...
Abstract interpretation... (1.29s)
Proving... (0.44s)                                                   OK
------------------------------------------------------------------------
Checking event 10/10...
Abstract interpretation... (0.86s)
Proving... (0.43s)                                                   OK
------------------------------------------------------------------------
                     No error was found in any event!
------------------------------------------------------------------------
Checking global events sequence...                                   OK
------------------------------------------------------------------------
                                 The program satisfies the specification!
------------------------------------------------------------------------
 Timing information
------------------------------------------------------------------------
Code generation ____________________________________________________ 0.01s
Abstract Interpretation ____________________________________________ 17.39s
Tasks generation ___________________________________________________ 0.38s
Tasks proof ________________________________________________________ 6.63s
Global _____________________________________________________________ 24.41s
------------------------------------------------------------------------
```

Fig. 15: Output of the tool on the nominal Carriage line control program.

5.2 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 15 depicts the output we obtain when running the analysis. In accordance with our strategy presented in Section 5.1, we consecutively infer invariants and then prove verification conditions for each pair (event, state), and for the initial *idle* state (S_0) of the timing chart. We observe that abstract interpretation is now much faster than during our first experiments [5], thanks to an optimization work on the Why3 plug-in. Those performances are now much closer to what one could expect of such a tool to be included in an industrial process.

5.3 Results on Incorrect Code

When the verification of a proof obligation fails for a so-called *faulty* event or state, our goal is to provide the most relevant information possible to a regular Ladder programmer, who might not be used to formal tools. We identified the three following interesting pieces of information:

1. The error location: at which event or which state the specification is violated,
2. The error reason: which output does not get its expected value at that location,
3. The error scenario: how could we get to this error. We propose to build such a scenario by mixing concrete values provided by counterexamples generated by Why3 (as described in Section 3.2), and abstract domains provided by abstract interpretation (as described in Section 3.1).

5.3.1 Error locations

With our current process, it is known at which pair of an event and a state an error occurs since the different pairs of an event and a state are treated separately. But we still need to distinguish if the error occurs in the event or in the state. As stated in Section 4.2, the fact that an event specification is satisfied is ensured by the proof of the initialisation of the invariant on outputs of the loop of the associated state, while the fact that the state specification is satisfied is ensured by the proof of preservation of that same invariant.

An easy way to distinguish those two cases consists of adding an additional assertion containing the same formula as the loop invariant just before the loop. Then, in order to distinguish which of the assertion or the invariant has failed to be proved, we use the *attributes* mechanism of Why3, which allows us to attach a label to a WhyML assertion (or invariant), this label being transported to verification conditions that are produced by the Weakest-Precondition Calculus and sent to solvers, Why3 finally attaching them to the potential counterexamples it returns for the concerned verification conditions.

We describe in Figure 16, how we insert those attributes in the WhyML modelling of event E_1 and state S_1.

```
scan(); (* Figure 9b *)
x0 := randomb();
assert { [@expl:Event_1] !y70 && not !y71 && not !y72 && not !y73 && not !y74 }

while (!x0 && not !x1 && not !x2 && !x3 && !x4) do
  invariant { [@expl:State_1]
              !y70 && not !y71 && not !y72 && not !y73 && not !y74 }
  scan(); (* Figure 9b *)
  x0 := randomb();
done
```

Fig. 16: WhyML modelling of event E_1 and state S_1 with location attributes.

```
scan(); (* Figure 9b *)
x0 := randomb();
assert { [@expl:Event_1 Y70:false] !y70 }
assert { [@expl:Event_1 Y71:true] not !y71 }
assert { [@expl:Event_1 Y72:true] not !y72 }
assert { [@expl:Event_1 Y73:true] not !y73 }
assert { [@expl:Event_1 Y74:true] not !y74 }

while (!x0 && not !x1 && not !x2 && !x3 && !x4) do
  invariant { [@expl:State_1 Y70:false] !y70 }
  invariant { [@expl:State_1 Y71:true] not !y71 }
  invariant { [@expl:State_1 Y72:true] not !y72 }
  invariant { [@expl:State_1 Y73:true] not !y73 }
  invariant { [@expl:State_1 Y74:true] not !y74 }
  scan(); (* Figure 9b *)
  x0 := randomb();
done
```

Fig. 17: WhyML modelling of event E_1 and state S_1 with location and reason attributes.

5.3.2 Error reasons

We want now to determine which output(s) is (are) concerned when a specification is violated, *i.e.*, an assertion or an invariant is found not to be provable. A simple way consists in separating each atomic formula in assertions and invariants and use the *attributes* mechanism as in the previous section. We describe in Figure 17, how we insert those attributes in the WhyML modelling of event E_1 and state S_1 (we directly use the opposite value as the expected one in the attribute, for printing convenience).

5.3.3 Error scenarios

As said in Section 5.1, when a proof obligation fails for a given event or state, we concatenate all the WhyML modellings of events and states until the faulty one, and send again this concatenation to Why3, which might give back a counterexample. The information we get then provides the values of (some of) the inputs,

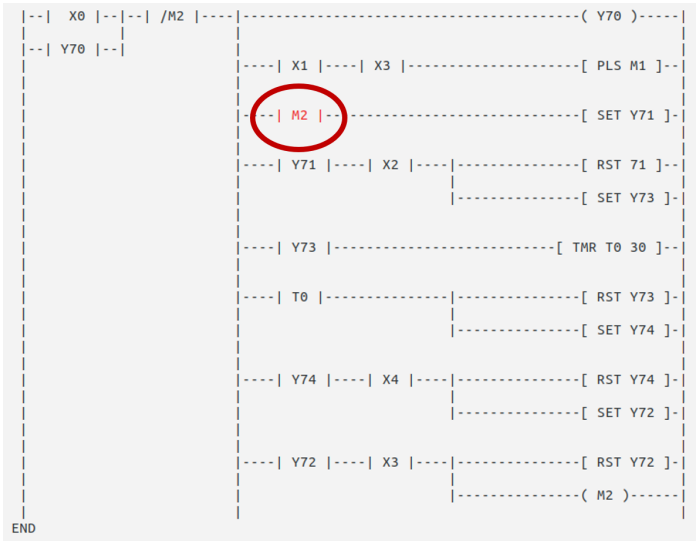


Fig. 18: First incorrect program.

outputs and internal devices at the beginning of each event until the error. We are therefore able to provide an execution trace concerning events until the error. But, 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, see Section 3), we do not have any information on the values taken by the devices during the states. We think that this lack of information concerning values of devices during 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 error (including the faulty event when applicable), the values of devices before the beginning of the scan of this event, obtained from the counterexample trace provided by Why3.
2. For each state that precedes the error, an over-approximation of domains of devices values, obtained by abstract interpretation.

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. Two of them are presented in this article. The corresponding Ladder diagrams are depicted in Figures 18 and 20. The modifications compared to the original code are encircled.

5.3.4 First incorrect program

The first modification of the Ladder program (Figure 18) corresponds to the use of the M2 internal memory device (instead of M1) in the contact front of rung 3. When trying to prove this program, our tool outputs an error location: event E_3,

```
------------------------------------------------------------------------
Verification condition 48 for Event_3 could not be proved:
Reason: Y71:false
------------------------------------------------------------------------
Error scenario:

Values of devices at event 1 scan beginning:
  M1 = false
  M2 = false
  X0 = true
  Y70 = false
  Y71 = false
  Y72 = false
  Y73 = false
  Y74 = false
  T0.current = 0
  T0.setting = 30

Values of devices during state 1:
  M1 ∈ {false; true}
  M2 = false
  X1 = false
  X2 = false
  X3 = true
  X4 = true
  Y70 = true
  T0.current = 0
  T0.setting = 30

Values of devices at event 2 scan beginning:
  M1 = false
  M2 = false
  X0 = false
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = false
  Y74 = false
  T0.current = 0
  T0.setting = 30

Values of devices during state 2:
  M1 ∈ {false; true}
  M2 = false
  X0 = false
  X2 = false
  X3 = true
  X4 = true
  Y70 = true
  T0.current = 0
  T0.setting = 30

Values of devices at event 3 scan beginning:
  M1 = false
  M2 = false
  X1 = true
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = false
  Y74 = false
  T0.current = 0
  T0.setting = 30
```

Fig. 19: Output of the tool for incorrect program 1.

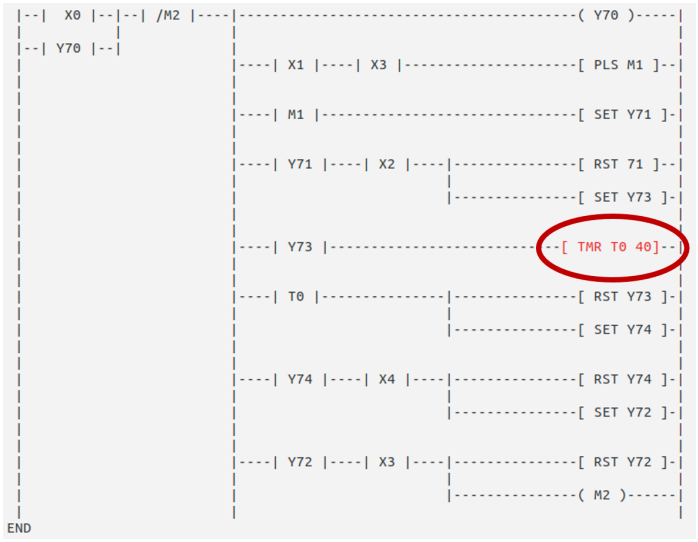


Fig. 20: Second incorrect program.

and an error reason: Y71 is false while it should be true. We also obtain an error scenario trace, which is depicted by Figure 19.

In this example, M2 is responsible of setting Y71 while M1 has this role in the nominal case. As M2 is supposed to be deactivated as long as the PLC runs, the consequence of this change is that Y71 is never activated. We observe that the value of M2 at the beginning of scan of the event 3 is equal to false, so there is no chance that Y71 is activated by the SET instruction. It indeed corresponds to a violation of event E_3 of the timing chart.

5.3.5 Second incorrect program

The second incorrect program is shown in Figure 20. 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 Figures 21 and 22.

The error scenario 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 state S_7, and reach 29 at the beginning of event E_8, instead of the expected value 39 (in which case it would have been incremented to 40 and that would have deactivated Y73 during event E_8 as specified).

Note that our tool outputs also a (similar) error scenario for the second violation at event E_8 that concerns the fact that Y74 is not activated.

```
------------------------------------------------------------------------
Verification condition 135 for Event_8 could not be proved:
Reason: Y73:true
------------------------------------------------------------------------
Error scenario:

(...)

Values of devices at event 5 scan beginning:
  (...)
  T0.current = 1
  T0.setting = 40

Values of devices during state 5:
  (...)
  T0.current ∈ [1; 29]
  T0.setting = 40

Values of devices at event 6 scan beginning:
  M1 = false
  X4 = false
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = true
  Y74 = false
  T0.current = 2
  T0.setting = 40

Values of devices during state 6:
  M1 = false
  X4 = false
  X2 = true
  X3 = false
  X4 = false
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = true
  Y74 = false
  T0.current ∈ [2; 29]
  T0.setting = 40
```

Fig. 21: Output of the tool for incorrect program 2 (part 1).

5.4 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 to a given timing chart specification. A weakness of this approach is that in some cases, the abstract values are imprecise compared to the concrete values. For example, in the error scenario for incorrect program 2 in Figures 21-22, it is said that the counter of the timer can take take all the values between 2 and 29 for state 6, while the concrete value given for the next event is 3. It means that the loop corresponding to state 6 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 [2;29] into [2;3], and even make explicit to the programmer that there is exactly one execution of

```
Values of devices at event 7 scan beginning:
  M1 = false
  X1 = false
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = true
  Y74 = false
  T0.current = 3
  T0.setting = 40

Values of devices during state 7:
  M1 = false
  X0 = false
  X1 = false
  X2 = true
  X3 = false
  X4 = false
  Y70 = true
  T0.current ∈ [3; 29]
  T0.setting = 40

Values of devices at event 8 scan beginning:
  M1 = false
  M2 = false
  Y70 = true
  Y71 = false
  Y72 = false
  Y73 = true
  Y74 = false
  T0.current = 29
  T0.setting = 40
```

Fig. 22: Output of the tool for incorrect program 2 (part 2).

the program for state 6. In an analog way, when errors happen during states, it would be interesting to provide an concrete execution trace inside the state until the error, instead of the imprecise abstract interpretation domain. This is kept for future work.

5.5 Graphical User Interface for Displaying Errors Scenarios

As seen in Section 5.3.3, we propose to the user, as an error scenario, the list of concrete values of devices at events beginnings (given by Why3's counterexamples) and the list of abstract domains of values during states (given by Why3's abstract interpretation), until the event or state in which the specification is violated. In order to help the user to navigate in such an error scenario, we developed a web-based graphical interface. This graphical interface highlights the error reason and location and allows to navigate in events and states execution values by clicking on the timing chart. For states, it prints the computed abstract domains of devices. For events, it allows to display the intermediate execution values of devices inside an event, in two graphical ways. First, it prints the Ladder program and colours in blue the active devices and locations in the program. Second, it also displays the evolution of devices values during the execution of the event as another timing chart. Those two views are synchronised via their two respective sliders. Finally, this interface allows to load the Ladder program and the timing chart, and to

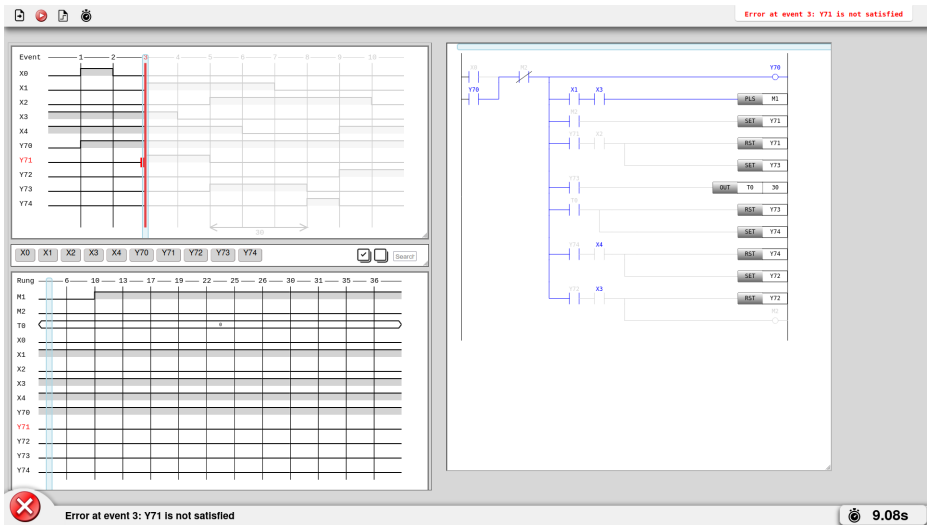


Fig. 23: Graphical User Interface: Error reason and location.

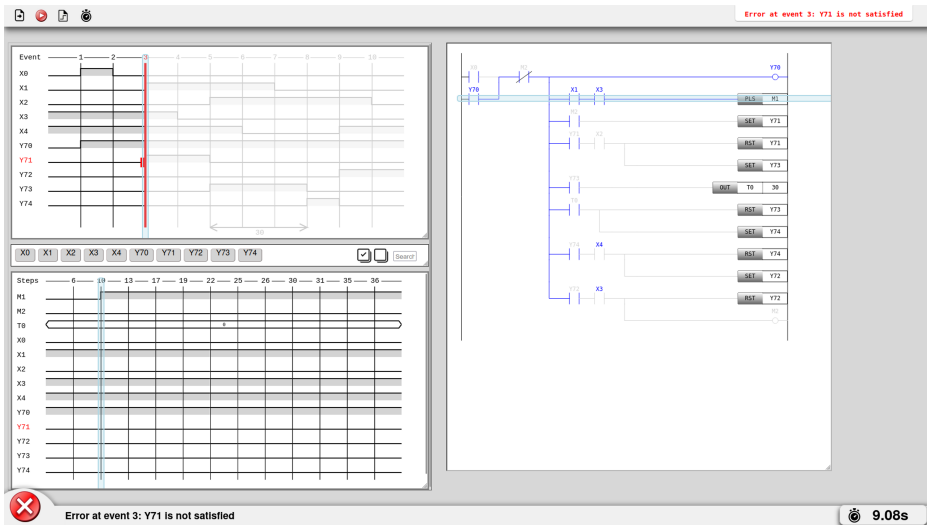


Fig. 24: Graphical User Interface: Sliders synchronisation for event concrete values.

launch the verification in a push-button and fully automated way, which is a key point for a smooth and harmless integration in industrial processes.

We describe the interface on the two incorrect programs of Section 5.3.4 and Section 5.3.5. Figure 23 shows the resulting interface when launched on first incorrect program. One can see the error reason and location highlighted in the top-right message ("`Error at Event_3: Y71 is not satisfied`"). The error reason and location are also highlighted in the top-left representation of the timing

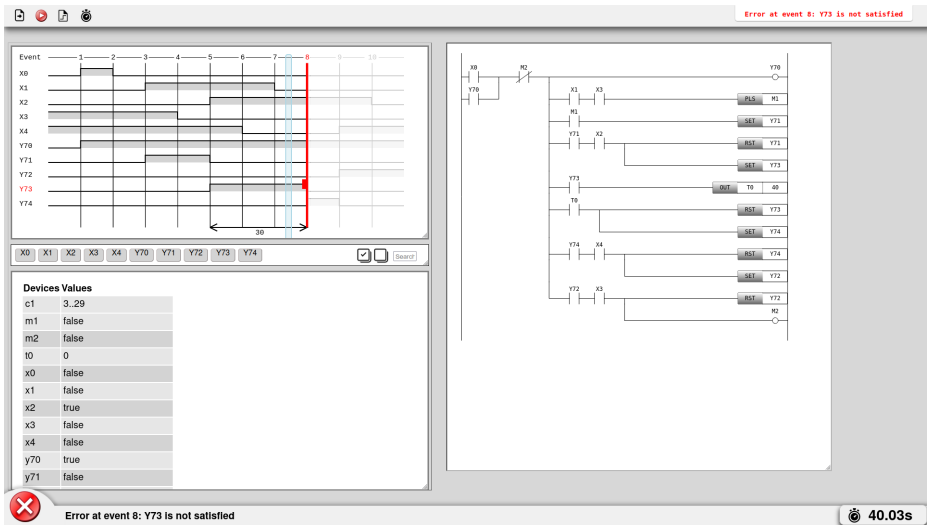


Fig. 25: Graphical User Interface: State abstract domains.

chart (event 3 and Y71 are highlighted in red). The faulty event (here E_3) is automatically selected by the slider in the top-left timing chart. This selection impacts what is displayed in the bottom-left and right sub-windows.

The right sub-window displays the program. When an event prior to the error is selected in the timing chart, it displays the execution trace for this event using colours (blue means activated and gray means deactivated). When a state is selected, it displays the whole program in black. The bottom-left sub-window displays execution values for devices. When a state prior to the error is selected, it displays the abstract domains as shown in Figure 25. When an event prior to the error is selected, it displays the execution trace for this event, as a timing chart, in a synchronised manner with the right sub-window, thanks to their respective sliders, as shown in Figure 24.

We believe that such a graphical interface would be very useful when debugging Ladder programs and that having such a fully automated prototype is a key point for convincing Ladder programmers of the usability of our solutions for being integrated in industrial processes.

5.6 Discussion on the Industrial Applicability

We demonstrated our work on a rather simple example controlling a carriage line. Actual industrial Ladder programs might be much bigger and composed of many of such sub-programs controlling sub-systems of a factory. However, we think that our work is of interest for formally verifying that all such sub-programs satisfy their timing-charts specifications, which is a mandatory milestone before a possible formal verification of a whole program w.r.t. a higher-level specification.

Timing charts are often used by PLC manufacturers for specifying the *Function Blocks* (FBs for short) they provide together with the PLCs they sell and their

companion programming environments. These FBs can be seen as standard library functions that can be used as elementary bricks to build more complex programs.

The performances of the tool we describe in this paper depend much more of the size of the given timing chart (the number of devices, events and states) than the size of the given program. Indeed, as we have seen in Figure 15, abstract interpretation takes much more time than translation to WhyML, VC generation and proving by back-end solvers. Moreover, we experimented in previous work [11] that those three phases (translation, VC generation and proving) are efficient (a few seconds) for programs whose sizes are about 50 times bigger than the Carriage line control example.

The timing chart of the Carriage line control example contains 10 devices and 11 events/states. This is very comparable to the timing charts we found in two sets of FBs that are shipped by Mitsubishi Electric with the programming environment GX Works3. The first set [24] is shipped with MELSEC iQ-F series PLCs. It contains 15 timing charts with average numbers of 11 devices and 7 events. The second set [25] is shipped with MELSEC iQ-R Series ("safety") PLCs. It contains 24 timing charts with average numbers of 11 devices and 10 events. This is very comparable to the running example of this paper. As one can see in MELSEC iQ-R safety FB reference [25], safety FBs might also be specified by *State Transition Diagrams* (STDs) that allow to specify several execution scenarios at once. We plan to extend our current work to handle such STDs as specifications.

6 Discussions, Related Work and Future Work

We presented a new method for verifying that a given Ladder code complies with an expected temporal behaviour expressed by a timing chart. The underlying verification procedure is fully automated, thanks to the loop invariant inference that avoids the need for the user to provide such invariants manually. On the one hand, when all the generated verification conditions are validated, one obtains a formal proof of conformity of the Ladder code with respect to the timing chart. The level of confidence in the proof is high, relying on Why3's VC generation and back-end solvers unsatisfiability checking. On the other hand, when the 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 error 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 4. It also relies on the soundness of the VC generation process of Why3 and the soundness of the back-end solvers unsatisfiability checking; both being external software components already widely used in other contexts. Regarding the 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 error scenarios, we have noticed that they are satisfactory on our case study, but we cannot guarantee that the generated scenarios are always valid ones. The potential sources of invalidity of counterexamples are well-known [12]: the loop invariants generated might be insufficient (inherent incompleteness of the VC generation) or the SMT solvers are unable to solve the generated VCs even if they are valid (inherent incompleteness of unsatisfiability checking in presence of quantified formulas, or more practically because the solvers are interrupted after a given time limit). There is on-going work in the Why3 development team to increase the trust into the validity of generated counterexamples [4] to avoid presenting a spurious counterexample to the user.

6.1 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.* [28] published a summary of those techniques. In 2016, Darvas *et al.* [13] 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 [17, 27] and PLC software [6] (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 [11], 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 [31], based on a translation from LTL formulas into annotations in the ACSL language [3]. 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 [15], 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.

6.2 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 (see end of Section 5.2), 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 5.3, 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* [4], 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 5.3.5, 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. Also, as said above, the counterexample built from the solvers might appear to be spurious, that is not illustrating into mistake in the given code. Beyond detecting that a counterexample is indeed spurious (which is still future work [4]), the question of how to give useful feedback in that case is open. Possible means include trying to use other solvers, or give them a larger time limit. In any case, at least the first problematic step in the timing chart is identified so as to guide a manual investigation, as a last resort.

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.

Acknowledgements We thank the anonymous reviewers for their constructive comments on earlier versions of this paper, and also thank the editors Alberto Lluch Lafuente and Anastasia Mavridou for their feedback and suggestions. All of them helped us to produce a greatly improved final version of this article.

References

1. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV – Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.
2. Lucas Baudin. Deductive verification with the help of abstract interpretation. Technical report, Univ Paris-Sud, 2017. URL: https://hal.inria.fr/hal-01634318.
3. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.16*, 2020. URL: https://frama-c.com/html/acsl.html.

4. Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. Explaining counterexamples with giant-step assertion checking. In *F-IDE – 6th Workshop on Formal Integrated Development Environments*, Electronic Proceedings in Theoretical Computer Science, 2021. URL: https://hal.inria.fr/hal-03217393, doi:10.4204/EPTCS.338.10.
5. Cláudio Belo Lourenço, Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, and Hiroaki Inoue. Automated verification of temporal properties of Ladder programs. In *FMICS – Formal Methods for Industrial Critical Systems*, volume 12863 of *Lecture Notes in Computer Science*, pages 21–38, 2021. URL: https://hal.inria.fr/hal-03281580, doi:10.1007/978-3-030-85248-1_2.
6. Sebastian Biallas, Stefan Kowalewski, Stefan Stattelmann, and Bastian Schlich. Efficient handling of states in abstract interpretation of industrial programmable logic controller code. In *12th International Workshop on Discrete Event Systems*, pages 400–405. IFAC, 2014.
7. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie – First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011. URL: http://hal.inria.fr/hal-00790310.
8. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.
9. W. Bolton. *Programmable Logic Controllers (Sixth Edition)*. Newnes, 2015. doi:10.1016/C2014-0-03884-1.
10. Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT – International Workshop on Satisfiability Modulo Theories*, 2018. URL: https://hal.inria.fr/hal-01960203.
11. Denis Cousineau, David Mentré, and Hiroaki Inoue. Automated deductive verification for ladder programming. In *F-IDE – Fifth Workshop on Formal Integrated Development Environments*, volume 310 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–12, 2019. doi:10.4204/EPTCS.310.2.
12. Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming*, 99:97–113, 2018. URL: https://hal.inria.fr/hal-01802488, doi:10.1016/j.jlamp.2018.05.003.
13. Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In *IFM – Integrated Formal Methods*, volume 9681 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2016. doi:10.1007/978-3-319-33693-0_32.
14. Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS – Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
15. Steven De Oliveira, Virgile Prévosto, and Sébastien Bardin. Au temps en emporte le C. In *JFLA – Vingt-sixièmes Journées Francophones des Langages Applicatifs*, 2015. URL: https://hal.inria.fr/hal-01099128.
16. Rainer Drath, Arndt Luder, Joern Peschke, and Lorenz Hundt. AutomationML – the glue for seamless automation engineering. In *ETFA – IEEE International Conference on Emerging Technologies and Factory Automation*, pages 616–623, 2008. doi:10.1109/ETFA.2008.4638461.
17. Ansgar Fehnker, Ralf Huuck, Bastian Schlich, and Michael Tapp. Automatic bug detection in microcontroller software by static program analysis. In *SOFSEM – Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2009. doi:10.1007/978-3-540-95891-8_26.
18. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV – 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007. URL: https://hal.inria.fr/inria-00270820v1, doi:10.1007/978-3-540-73368-3_21.
19. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *ESOP – 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. URL: http://hal.inria.fr/hal-00789533.

20. Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In *ISoLA – 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142. Springer, 2020. See also http://why3.lri.fr/isola-2020/. URL: https://hal.inria.fr/hal-02696246.
21. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV – Computer Aided Verification*, pages 661–667. Springer, 2009.
22. Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In *ISoLA – 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478. Springer, 2016. URL: https://hal.inria.fr/hal-01344110, doi:10.1007/978-3-319-47166-2_32.
23. John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.
24. Mitsubishi Electric Corporation. MELSEC iQ-F FX5 CPU Module Function Block Reference (for GX Works3). https://dl.mitsubishielectric.com/dl/fa/document/manual/plcf/jy997d62701/jy997d62701j.pdf, 2016. [Online; accessed 14-June-2022].
25. Mitsubishi Electric Corporation. MELSEC iQ-R Safety Function Block Reference (for GX Works3). https://dl.mitsubishielectric.com/dl/fa/document/manual/plc/bcn-p5999-0815/bcnp59990815c.pdf, 2016. [Online; accessed 14-June-2022].
26. 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].
27. T. Nguyen, T. Aoki, T. Tomita, and J. Endo. Integrating static program analysis tools for verifying cautions of microcontroller. In *APSEC – Asia-Pacific Software Engineering Conference*, pages 86–93, 2019. doi:10.1109/APSEC48747.2019.00021.
28. Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Ünver. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling*, 15:1–24, 2014. doi:10.1007/s10270-014-0448-7.
29. R. Ramanathan. The IEC 61131-3 programming languages features for industrial control systems. In *WAC – World Automation Congress*, pages 598–603, 2014. doi:10.1109/WAC.2014.6936062.
30. Arnaud Roques. PlantUML standard library. https://plantuml.com/stdlib, 2009. [Online; accessed 24-March-2021].
31. Nicolas Stouls and Julien Groslambert. Vérification de propriétés LTL sur des programmes C par génération d'annotations. Research report, 2011. URL: https://hal.inria.fr/inria-00568947.