A Methodological Guide for the Validation of Logic Modelling of Ladder Instructions

Denis Cousineau, Hiroaki Inoue, Claude Marché, David Mentré

A Methodological Guide for the Validation of Logic Modelling of Ladder Instructions

Denis Cousineau, Hiroaki Inoue, Claude Marché, David Mentré

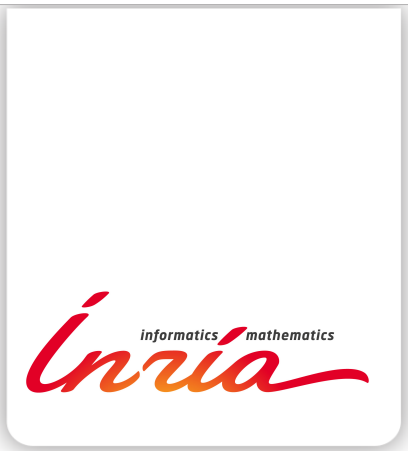
TECHNICAL REPORT

N° 0522

March 2024

Project-Team Toccata

ISSN 0249-0803 ISRN INRIA/RT--0522--FR+ENG



A Methodological Guide for the Validation of Logic Modelling of Ladder Instructions*

Denis Cousineau†, Hiroaki Inoue†, Claude Marché‡, David Mentré†

Project-Team Toccata

Technical Report n° 0522 — March 2024 — 22 pages

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 a previous work [3], we proposed a method for verifying that a given Ladder program conforms to an expected behaviour expressed by a *timing chart*, describing a scenario of execution. This method relies on a modelling of Ladder programs in WhyML, the language of the Why3 environment for deductive program verification. In this approach, the WhyML modelling of individual Ladder instructions has to be trusted.
This report proposes a methodology to increase the trust in the WhyML modelling of Ladder instructions. Our approach relies on a comparison of the execution of Ladder programs with an execution by Why3 of a simulation of the translated program. With this technique, we have been able to validate our modelling of Ladder instructions, and also discover and fix a subtle bug in the modelling of one particular instruction.

Key-words: Ladder Language for PLCs, Formal Specification, Why3 Environment for Deductive Verification, Model Validation

* This work has been partially supported by the bilateral contract ProofInUse-MERCE between Inria team Toccata and Mitsubishi Electric R&D Centre Europe, Rennes.
† Mitsubishi Electric R&D Centre Europe, Rennes, France
‡ Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Un guide méthodologique pour la validation de modèles logiques des instructions Ladder

Résumé : Les automates programmables industriels (PLC en anglais) sont des dispositifs numériques industriels utilisés pour contrôler les processus de fabrication automatisés, tels que les lignes d'assemblage. Le langage Ladder est un langage de programmation utilisé pour développer des logiciels pour de tels dispositifs. Dans un travail précédent [3], nous avons proposé une méthode pour vérifier qu'un programme Ladder donné est conforme à un comportement attendu exprimé par un *diagramme de temps*, décrivant un scénario d'exécution. Cette méthode s'appuie sur une modélisation des programmes Ladder en WhyML, le langage de l'environnement Why3 pour la vérification déductive des programmes. Cette méthode s'appuie sur une modélisation des instructions individuelles Ladder, à laquelle il faut faire confiance.

Ce rapport propose une méthodologie pour accroître la confiance dans la modélisation WhyML des instructions Ladder. Notre approche repose sur une comparaison des exécutions d'un programme Ladder avec une exécution par Why3 d'une simulation du programme traduit. Grâce à cette technique, nous avons pu valider notre modélisation des instructions Ladder, et également découvrir et corriger un bug subtil dans la modélisation d'une instruction particulière.

Mots-clés : langage Ladder pour les PLC, Spécification formelle, environnement Why3 pour la vérification déductive, Validation de modèle

Ces recherches ont été partiellement financées par le contrat bilatéral ProofInUse-MERCE entre l'équipe Inria Toccata et Mitsubishi Electric R&D Centre Europe, à Rennes.

Contents

List of Figures

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 [15] among other languages [6] 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. For this purpose, we proposed to apply deductive verification platform Why3 to Ladder logic [3]. In this study, we have given a formal model of Ladder logic semantics using Why3, and built a software tool to check that a given Ladder program conforms to expected scenarios of execution expressed as timing charts.

One problem is how to validate that the formal model is conforming to actual Ladder semantics. Unfortunately, unlike C language, we don't have common semantics of Ladder logic. Although there is IEC 61131-3 standard to describe Ladder logic, it doesn't refer about detailed semantics. Therefore, we need to rely on each implementation of Ladder logic on which we want to integrate our verification methods: in our case, GX Works3 of Mitsubishi Electric Corporation. However, it is a complex task to give precise formal model of the semantics even if we can read its internal implementation.

In this report, we study a lightweight approach to validate our formal model according to execution logs of Ladder logic. In Section 2 we give an overview of Ladder programming and especially present how one could obtain traces of execution of a given Ladder program, under the form of execution logs. In Section 3 we present the Why3 environment and its WhyML language, and we summarise how Ladder programs are encoded into WhyML. The Section 4 is the main contribution of this report, presenting our methodology of validation of the WhyML modelling. We present the results obtained on some instructions, in particular on timer instructions in which our methodology allowed us to expose a small discrepancy between the Ladder execution and the WhyML model. We conclude in Section 5 with some plans for future work.

2 Overview of 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, 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 or in a parallel way (the obtained value is then the conjunction, resp. the disjunction, of the two contact values). 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, there 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*.

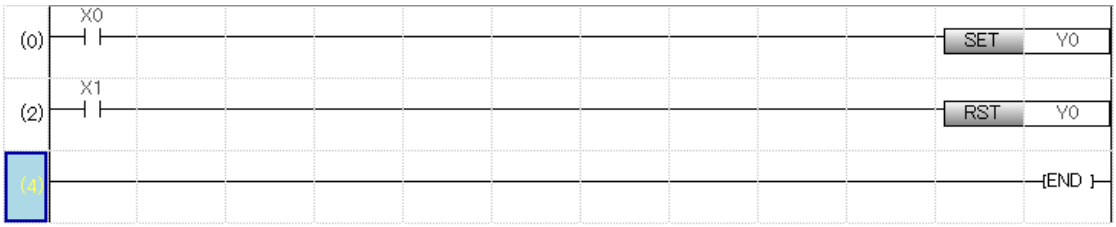


Figure 1: A simple Ladder program involving instructions SET and RST.

2.1 Illustrative Example of a Ladder Program

A simple example is depicted in Figure 1, which includes Ladder instructions SET and RST. This program has two inputs X0 and X1, and one output Y0. 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. In the example, Y0 is activated when X0 is activated. The RST instruction is the opposite: the device argument is deactivated when the instruction's input is activated. That is, Y0 is deactivated when X1 is activated.

2.2 How to Obtain Execution Logs

To obtain a sample execution log of a given Ladder program, we use a data logging tool. We can find several tools such as ones depending on OPC Unified Architecture standard, or vendor-specific ones. Here, we use the vendor-specific software of Mitsubishi Electric Corporation, called GX LogViewer.

Using this tool, one can run a given Ladder program, in some interactive manner. In particular one can interactively activate, or deactivate, inputs. The tool provides a trace of execution, under the form of a CSV file giving the values of each inputs and outputs at each scan. Figure 2 shows an example of an execution log of the Ladder program of Figure 1. INDEX means the number of scan from the beginning of data logging. X0, X1 and Y0 indicate device values at the end of each scan execution. We use vertical dots to abbreviate rows of identical values as the previous row. At first, X0 remains deactivated (keeps 0) until the scan 7146, the time when X0 is activated (set to 1), causing Y0 being activated somehow immediately. Later on X0 is deactivated (at scan 9303), but Y0 remains activated. When X1 becomes activated (at scan 14618), Y0 is deactivated. If both X0 and X1 are activated (as at scan 26062), Y0 becomes deactivated since a Ladder program is executed from the top rung to the bottom rung.

3 Modelling Ladder Instructions using WhyML

Why3 is a generic environment for deductive program verification, providing the language WhyML for specification and programming [12, 5, 13]. 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 [11, 14, 10]. The specification component of WhyML [4], 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 [8], CVC4 [1] and Z3 [9].

INDEX	X0	X1	Y0
1	0	0	0
2	0	0	0
⋮	⋮	⋮	⋮
7145	0	0	0
7146	1	0	1
7147	1	0	1
⋮	⋮	⋮	⋮
9302	1	0	1
9303	0	0	1
9304	0	0	1
⋮	⋮	⋮	⋮
14617	0	0	1
14618	0	1	0
14619	0	1	0
⋮	⋮	⋮	⋮
16915	0	0	0
⋮	⋮	⋮	⋮
22381	1	0	1
⋮	⋮	⋮	⋮
26062	1	1	0
⋮	⋮	⋮	⋮

Figure 2: An execution log of the SET-RST Ladder program

```
1  module M
2
3  use int.Int
4  use ref.Ref
5
6  (** global variable ‘s‘ storing partial sums *)
7  val s : ref int
8
9  (** toy function ‘f‘, when Boolean argument is False then the sum is
10     reset to zero, otherwise it is incremented by ‘x‘ *)
11 let f (x:int) (b:bool) : unit
12   writes { s }
13   ensures { b → !s = old !s + x }
14   ensures { not b → !s = 0 }
15 =
16   if b then s := !s + x else s := 0
17
18 (** sample test program *)
19 let test () =
20   f 0 False;
21   assert { !s = 0 };
22   f 13 True;
23   assert { !s = 13 };
24   f 21 True;
25   assert { !s = 34 };
26   f (!s) True;
27   assert { !s = 68 };
28   f 1 False;
29   assert { !s = 69 } (* this assertion is intentionally not correct *)
30
31 end (* module M *)
```

Figure 3: Toy example of a WhyML program with a formal contract.

3.1 Overview of the WhyML Language

We illustrate the essential Why3 features on the toy WhyML program presented in Figure 3. This code involves a global variable `s` of type integer, that is a mathematical, unbounded integer in WhyML. The toy function `f` takes two arguments, an integer `x` and a Boolean `b`. It is equipped with a *formal contract*: first a *frame clause*, introduced by keyword `writes` on line 12, stating that the function can modify the global variable `s`, and nothing else. This contract also involves two postconditions, introduced by keyword `ensures` on lines 13-14, stating respectively that at exit, if `b` is true then `s` is incremented by `x`, and that if `b` is false, then `s` is reset to zero. Notice the WhyML syntax for mutable variables, inspired by ML, requiring to write an exclamation mark to access their values. The function `test` takes no argument at all. The body of `test` involves several calls to the previous function `f`. Notice the notation of call which is *curryfied*, meaning that argument are juxtaposed after the function name, without any parentheses, as usual in ML syntax. The body of `test` also involves another kind of formal annotation, namely code

assertions introduced by keyword `assert`. Such an assertion specifies that the given condition should hold when execution reaches this program point.

Given such an annotated code, the main functionality of Why3 core engine is to generate *Verification Conditions* (VCs). These are mathematical formulas expressing that the code conforms to its formal annotations. In the example, VCs are generated to show that the body of function `f` conforms to its contract, and that the body of function `test` respects the given assertions. In this example, all the generated formulas are proved correct by SMT solvers, except the formula for the last assertion, which is expected since this assertion is indeed invalid.

If the main functionality of Why3 is the generation of VCs, it is not the only one. The one we use in this report is the ability to execute code, as described below. This execution includes evaluation of logic annotations if possible. This form of execution is classically called *Runtime Assertion Checking* and abbreviated as RAC.

3.2 Runtime Assertion Checking of WhyML Code

Given a WhyML program, it is possible to execute any expression involving the functions defined in the program. The Why3 command to do so has the following shape.

```
> why3 execute file.mlw --use modulename "expr"
```

where `file.mlw` is the file name where the code is, *modulename* the name of the module to be considered in that file, and *expr* is the expression to execute. For example, on the toy example of Figure 3, one can run the test function using

```
> why3 execute toy.mlw --use M "test␣()"
```

and the result is as follows.

```
result: () = ()
globals: s -> {contents= 0}
```

Execution simply returns the result of the expression evaluated, which is here `()`, and the final state of the global variables, here that `s` contains 0.

The execution above did not take formal annotations into account at all, these are simply ignored. It is yet possible to execute a program while checking the annotations: it is called *Runtime Assertion Checking* and abbreviated as RAC. Such an execution is done by Why3 when the additional option `--rac` is passed on the command line, as follows.

```
> why3 execute file.mlw --use modulename --rac "expr"
```

With our toy example, the RAC gives the following output.

```
> why3 execute toy.mlw --use M --rac "test␣()"
Assertion failed at "toy.mlw", line 29, characters 11-18
- Term: !s = 69
- Variables: s -> {contents= 0}
```

Unlike before, an assertion failure is reported. It is identified on line 29, the offending condition is printed, and as before the current state of global variable is displayed.

Generally speaking, the Why3 RAC is thus able to expose non-conformity of code with respect to its formal annotations. As is, this process is a useful complementary tool with respect to the VC generation: proving all the VCs formally proves that a program conforms to its annotations on any inputs, whereas RAC is able to expose a non-conformity. Indeed, Becker *et al.* [2] used this ability to identify a run of the program exposing a violation can be used by Why3 itself to validate and categorise potential counterexamples returned by SMT solvers. We don't use this sophisticated feature in our validation methodology here, but we directly make use of the RAC.

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

let rst (input : bool) (device : ref bool) : unit
  writes { device }
  ensures { !device = (andb (notb input) (old !device)) }
= if input then device := False
```

Figure 4: WhyML functions encoding the Ladder instructions SET and RST. The function `set` encodes the SET instruction. It takes a parameter `input` denoting its input device and a parameter `device` denoting its output device. The output device parameter is a reference so as to be modifiable by the call to `set`. That function is given a contract specifying a `writes` clause stating that `device` can be modified, and an `ensures` clauses stating a post-condition: the new value of `device` is equal to the Boolean or of the input and its former value. The function `set` is also given a body, stating its effect operationally: if the input is on then the output is on, otherwise it is unchanged. The function `rst` encodes the RST instructions in an analogous fashion.

3.3 WhyML Encoding of Ladder Instructions

In our paper on verification of temporal properties of Ladder programs [3], we proposed an encoding of Ladder programs into WhyML code. Roughly speaking, each basic Ladder instructions, such as SET and RST seen above, are modelled using WhyML functions taking as parameters the input device and as *mutable* parameters the output devices.

As an example, Figure 4 displays the two WhyML functions encoding SET and RST. These two functions can be shown conforming to their contracts using the VC generation process described in Section 3.1 above. The proofs are done easily using SMT solvers.

A scan of the simple Ladder program given on Figure 1 is then encoded as another WhyML function `scan` as follows.

```
val x0 : ref bool
val x1 : ref bool
val y0 : ref bool

let scan () =
  set (!x0) (y0);
  rst (!x1) (y0)
```

An execution of that program can be done by calling `scan()` an arbitrary number of times, interleaved with modifications of the inputs. An example is as follows.

```
let test () =
  x0 := False; x1 := False;
  scan();
  let r1 = !y0 in x0 := True;
  scan();
  let r2 = !y0 in (r1,r2)
```

and the result is

```
> why3 execute sample_set_rst.mlw --use M --rac "test␣()"
result: (bool, bool) = (false, true)
globals: x0 -> {contents= true}, x1 -> {contents= false},
  y0 -> {contents= true}
```

As expected the output Y1 after the first scan is off, and is on after the second scan.

4 Validation Methodology by Comparison of Execution Traces

With the concepts and tools described in previous sections, we are now ready to describe our methodology for validating the WhyML modelling of Ladder instructions.

4.1 Overview of the Validation Methodology

Here is the flow of our validation method.

1. Design a Ladder program that exercises the Ladder instruction whose WhyML modelling is to be validated.
2. Run that Ladder program in GX Log Viewer and get one (or several) execution trace(s).
3. Translate each execution trace into a timing chart.
4. Generate a WhyML program which executes the Ladder program according to the timing charts, with assertions.
5. Execute the WhyML program with Runtime Assertion Checking.

A raw execution trace gotten in step 2 has a lot of duplicate events since they record results of every scan, as seen for example on Figure 2. Therefore, in step 3, we extract essential events by removing series of duplicated events. The result is expressed under the form of a timing chart, which describes the evolution of inputs and outputs over time. Then, in step 4, we reuse our tools developed for verification of temporal properties of Ladder programs [3] and generate a WhyML program simulating the execution of the original Ladder program according to the timing chart. This process embeds assertions in the WhyML code, which check the execution results (*i.e.* the values of outputs) are conforming to the timing chart. Finally, in step 5 of the process, we validate the Ladder program using Why3 runtime assertion checker: if any assertion violation is signaled, then an issue in the modelling has been exposed. Otherwise, without any assertion violation, the validation is passed.

In our previous work [3], we regarded a timing chart as a list of events and states. In this validation process, we ignore states since our main objective is to validate each instruction.

4.2 First Example: Validating the SET and RST modelling

We first illustrate our methodology by attempting to validate the modelling of SET and RST, as already shown in Figure 4. For the step 1 of our process, we use the simple SET-RST program of Figure 1. For the step 2, we already have shown a log of an execution trace, in Figure 2. By removing the duplicated consecutive events, we extract the events shown in Figure 5.

For step 4 of the process, we thus write a WhyML program simulating Ladder program according to the timing chart. The result is displayed in Figure 6. The `main` function is a program which executes according to the given timing chart. It is generated mechanically as explained in Section 4.1.

Executing this WhyML code with runtime assertion checking is done as follows.

INDEX	X0	X1	Y0
1	0	0	0
7146	1	0	1
9303	0	0	1
14618	0	1	0
16915	0	0	0
22381	1	0	1
26062	1	1	0

Figure 5: Extracted events from the execution log of SET-RST Ladder program

```
val y0 : ref bool

let scan (x0: bool)(x1: bool) =
    set (x0) (y0);
    rst (x1) (y0)

let main () =
    scan(false)(false); assert{not !y0};
    scan(true)(false);  assert{!y0};
    scan(false)(false); assert{!y0};
    scan(false)(true);  assert{not !y0};
    scan(false)(false); assert{not !y0};
    scan(true)(false);  assert{!y0};
    scan(true)(true);   assert{not !y0};
```

Figure 6: A WhyML program simulating the execution of the SET-RST test under the timing events given in Figure 5.

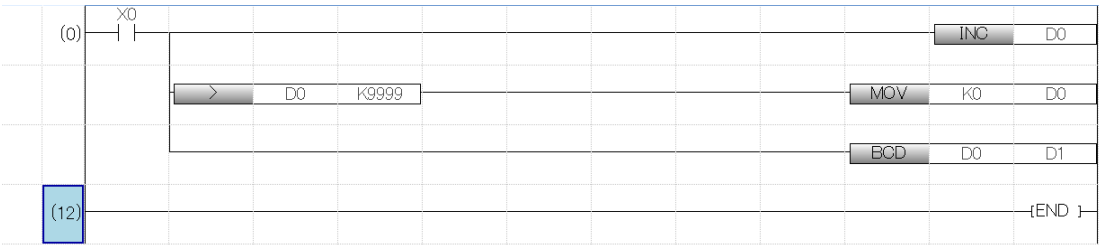


Figure 7: Ladder program for testing instruction BCD. When X0 becomes true, D0 is incremented, or (because of MOV, where K0 denoting the constant 0) becomes 0 if it becomes larger than 9999. The BCD-converted value of D0 is stored into D1. Remember that rungs are executed from top to bottom.

```
> why3 execute set_rst.mlw --use=Validation --rac "main␣()"
result: () = ()
globals: x0 -> {contents= true}, x1 -> {contents= true},
  y0 -> {contents= false}
```

Since we don't get any assertion violations, the WhyML modelling of SET and RST is validated by this test. See the concluding section for a discussion about the coverage of the validation.

4.3 Second Example: Binary Coded Decimal

Our second example is slightly more involved than the basic SET and RST instructions. It involves the instruction BCD, for Binary Coded Decimal. Additionally, we are also going to test two other simple instructions: INC and MOV. Our sample program involving those three instructions is shown in Figure 7. The INC instruction increments the value of its device argument D0. The MOV instruction stores a constant into its output device. The BCD instruction converts a 16 bits integer into a 16 bits Binary-Coded Decimal integer: the 16-bits BCD format represents 4 decimal digits, using 4 bits to represent each of the 4 digits. For example, it converts the number 1234 (in decimal) in the number 4660 (in decimal) which is 0x1234 in hexadecimal. It is typically used for display purpose: a 16-bit number lower than 10000 can be displayed by converting it with BCD and outputting each of the four 4-bits sub-bitvectors into say a four-digits 7-segment LCD displayer.

By running this program into GX Log Viewer, we get the execution log shown in Figure 8. The activation device X0 was activated. The number 39321 corresponds to 0x9999 in hexadecimal. At first, X0 remains 0 (false) until the scan 8389. Then, X0 becomes 1 (true) in scan 8340, causing D0 and D1 to become 1. While X0 remains true, the value of D0 is incremented until 9999 and D1 is changed accordingly.

The WhyML modellings of instructions INC, MOV and BCD are shown in Figure 9. Following our validation process, using the test Ladder program of Figure 7, and the timing chart corresponding to the sequence of events of Figure 8, we automatically generate the following WhyML code. First the encoding of one scan as follows.

```
let scan (x0: bool) =
  let common_1 = (x0) in
  inc_16u (common_1) (d0);
  let common_2 = (9999 < (!d0)) in
  mov ((common_1 && common_2)) (0) (d0);
  bcd (common_1) ((!d0)) (d1)
```

INDEX	X0	D0	D1
1	0	0	0
2	0	0	0
⋮	⋮	⋮	⋮
8389	0	0	0
8390	1	1	1
8391	1	2	2
8392	1	3	3
⋮	⋮	⋮	⋮
18387	1	9998	39320
18388	1	9999	39321
18389	1	0	0
18390	1	1	1
⋮	⋮	⋮	⋮

Figure 8: Extracted events from the execution log of BCD program

```
let mov (input : bool) (src : int) (tgt : ref int) : unit
  writes { tgt }
  ensures { (input ∧ !tgt = src) ∨ (not input ∧ !tgt = old !tgt) }
= if input then tgt := src

let inc_16u (input : bool) (dev : ref int) : unit
  writes { dev }
  ensures {    (not input ∧ !dev = old !dev)
            ∨ (input ∧ ((old !dev = 65535 ∧ !dev = 0) ∨ !dev = old !dev + 1)) }
= if input then dev := if !dev = 65535 then 0 else !dev+1

use int.EuclideanDivision

function bcd_compute_f (src : int) : int =
  let dig1 = div src 1000 in
  let r1 = mod src 1000 in
  let dig2 = div r1 100 in
  let r2 = mod r1 100 in
  let dig3 = div r2 10 in
  let dig4 = mod r2 10 in
  dig1*4096 + dig2*256 + dig3*16 + dig4

let bcd_compute_f_let (src : int) : int
  requires { 0 <= src ∧ src <= 9999 }
  returns { ret → ret = bcd_compute_f src }
= let dig1 = div src 1000 in
  let r1 = mod src 1000 in
  let dig2 = div r1 100 in
  let r2 = mod r1 100 in
  let dig3 = div r2 10 in
  let dig4 = mod r2 10 in
  dig1*4096 + dig2*256 + dig3*16 + dig4

let bcd (input : bool) (src : int) (tgt : ref int) : unit
  writes { tgt }
  ensures { input → !tgt  = bcd_compute_f src }
  ensures { not input → !tgt = old !tgt }
= if input then tgt := bcd_compute_f_let src
```

Figure 9: The WhyML modeling of instructions MOV, INC and BCD.

and then the encoding of the test program as follows.

```
let main () =
  scan(false); assert{!d0 = 0 && !d1 = 0};
  scan(true);  assert{!d0 = 1 && !d1 = 1};
  scan(true);  assert{!d0 = 2 && !d1 = 2};
  ...
  scan(true);  assert{!d0 = 1000 && !d1 = 4096};
  ...
  scan(true);  assert{!d0 = 9999 && !d1 = 39321};
  scan(true);  assert{!d0 = 0 && !d1 = 0};
  scan(true);  assert{!d0 = 1 && !d1 = 1};
  scan(false); assert{!d0 = 0 && !d1 = 0};
```

It contains around 10000 calls to the `scan` function. Executing this code appeared to be impossible for the runtime assertion checker of Why3. Indeed, an attempt to run only a few hundreds scans only gave the following results.

Number of scans	running time in seconds
100	6
200	25
300	49
400	82
500	121

The numbers in this table show that the time of execution is superlinear (but not as bad as quadratic) and globally very slow: even for an interpreter, executing 500 lines of code should not take 2 minutes. A workaround was to split the execution into smaller steps: we built one separate test for each sequence of 100 scans, as follows.

```
let main0 ()
  =
  d0 := 0;
  scan(false); assert{!d0 = 0 && !d1 = 0};
  scan(true);  assert{!d0 = 1 && !d1 = 1};
  scan(true);  assert{!d0 = 2 && !d1 = 2};
  ...
  scan(true);  assert{!d0 = 100 && !d1 = 256};

let main1 ()
  =
  d0 := 100;
  scan(true);  assert{!d0 = 101 && !d1 = 257};
  ...
  scan(true);  assert{!d0 = 200 && !d1 = 512};
```

Using this trick, we could validate the execution and thus validate the WhyML modellings of BCD (together with INC and MOV).

4.4 Third Example: Timer Instructions

Our third example involves a Ladder *timer*. Our sample Ladder code in Figure 10 illustrating how such a timer behaves.

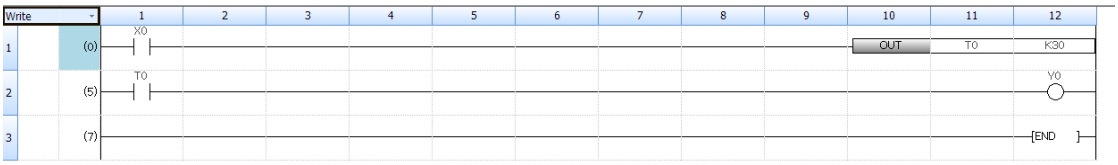


Figure 10: An example of a Ladder program involving a timer, here called T0. The *timer coil* defines 30 as its threshold (notation K30 means "constant value 30"). It means that after 30 deci-seconds in which the timer coil X0 for T0 is activated, the *timer contact* for T0 is activated and remains activated 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.

INDEX	X0	Y0	T0
1	0	0	0
2	0	0	0
⋮	⋮	⋮	⋮
6612	1	0	0
6613	1	0	0
6614	1	0	0
⋮	⋮	⋮	⋮
6787	1	0	1
6788	1	0	1
6789	1	0	1
⋮	⋮	⋮	⋮
7099	1	0	2
7100	1	0	2
⋮	⋮	⋮	⋮
15841	1	0	29
15842	1	1	30
15843	1	1	30
⋮	⋮	⋮	⋮
29738	1	1	30
29739	0	0	0
29740	0	0	0
⋮	⋮	⋮	⋮

Figure 11: Execution log of the Ladder program with a timer. Remark that in this simulation we kept the activation device X0 on for all the execution of the program. We could also try another simulation where we release X0 during the counting.

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

let timer_coil (input : bool) (t : timer) (v : int) : unit
  writes { t }
  ensures {  t.setting = v
           ∧ (    (input ∧ t.current = (old t.current) + 1)
                ∨ ((not input) ∧ t.current = 0)) }
=
  if input 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
```

Figure 12: WhyML modelling of Ladder timers, initial version.

We again used GX Log Viewer to extract one execution log, shown in Figure 11. The activation input X0 was pressed at index 6612. The value of T0 remained 0 for a while until index 6787 when it got to 1. This number of scans roughly corresponds to 0.1 second. The timer is then incremented regularly at each tenth of a second. At index 15842, the timer reached the value 30 for the first time and Y0 is activated at the same scan. Everything stayed unchanged until index 29739 where X0 was released, and the timer got back to 0.

Our WhyML modelling of timers is given in Figure 12. `timer` is a record type with two fields `current` and `setting`. The field `setting` is for holding the timer threshold, say 30 in the example, while `current` is it current value. the WhyML function `timer_coil` is the one that models a timer such as the first rung of the program of Figure 10. The function `timer_contact` is a Boolean function telling whether the given timer has reached its threshold.

The WhyML test program obtained from the Ladder program of Figure 10 and the log of Figure 11 as as follows.

```
1  val y0 : ref bool
2  val t0 : timer
3
4  let scan (x0 : bool) =
5    let common_5 = (x0) in
6    timer_coil (common_5) (t0) (30);
7    let common_7 = timer_contact (t0) in
8    y0 := common_7
9
10 let main () =
11   y0 := false; t0.current ← 0;  t0.setting ← 30;
12   scan(false); assert{t0.current = 0 && not !y0};
13   scan(true); assert{t0.current = 0 && not !y0};
```

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

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

Figure 13: WhyML modelling of timers, fixed version.

```
14   scan(true); assert{t0.current = 1 && not !y0};
15   scan(true); assert{t0.current = 2 && not !y0};
16   ...
17   scan(true); assert{t0.current = 29 && not !y0};
18   scan(true); assert{t0.current = 30 && !y0};
19   scan(false); assert{t0.current = 0 && not !y0};
```

The execution of this WhyML program is as follows.

```
> why3 execute --rac timer_error.mlw --use Validation "main ()"
Assertion failed at "timer_error.mlw", line 46, characters 21-46
- Term: t0.current = 0 && not !y0 = True
- Variables: t0 -> {current= 1; setting= 30}, y0 -> {contents= false}
```

An assertion violation is signaled. The line of the violation, 46, indeed corresponds to line 13 in the listing of the WhyML code above, which is abbreviated. It means the assertion after the second scan is not valid. Indeed, the assertion asks to check that `t0.current` is 0 but during the WhyML execution `t0.current` gets value 1. This is because that in our modelling: T0 counts from 1 when X0 becomes true. This is different from the log of Figure 11, where the counter value remains 0 for a tenth of a second. In other words, our validation methodology has been able here to reveal an error in our modelling.

Fixing the WhyML modelling of timers A way to fix this error is to introduce an additional field in timers, indicating whether the timer is counting or not. The result is shown on Figure 13. By repeating the process of our methodology, we are this time able to validate this new modelling.

5 Conclusions and Future Work

We presented a methodology for checking the validity of the WhyML modelling of Ladder instructions, as it is used in our work on verification of Ladder programs [3]. This validation methodology is partly mechanised: a WhyML program is automatically produced from a Ladder program and an execution log, and is executed using the runtime assertion checker of Why3. Any violation of an assertion reveals an error in the modelling. We experimented this methodology on several Ladder instructions including non trivial instructions such as Binary-Coded-Decimal conversion, and timers. An evident benefit of our experiments is that we discovered a subtle mistake in our initial modelling of timers, that we could fix and recheck.

We identified some drawbacks, the main one being the poor scalability of Why3 runtime assertion checker.

Future Work Work for the near future is to improve our tooling so as to make the methodology even more automated. Such improved automation is required to enlarge our coverage of Ladder instructions, which is for now limited to a small subset.

A mid-term future work is to improve the runtime assertion checker of Why3. There is no fundamental reason why its running time is so poor, so there is room for improvement. Also, there should be a way to validate a simulation using a purely concrete execution of WhyML code, by replacing assertions with some form of exceptions to raise when a condition does not hold. We already know though that the current implementation of the WhyML interpreter suffers from some weaknesses, such as sometimes the values of variables are seen as "undefined" although it should not be the case. Moreover, it should be noted that improving the implementation of the WhyML interpreter is likely to benefit to other applications, such as the checking and categorisation of counterexamples [2].

Longer-term future work include several topics. A first topic is to find a way to measure the coverage of the validation of Ladder instructions with our methodology. Indeed, as noted before, if a simulation does not reveal any assertion violation, it does not guarantee that the modelling is correct, because it depends whether the considered test program covers the different possible behaviours of the Ladder instruction under consideration.

Another long-term topic is the possibility of using an alternative validation method. Indeed, in a near future we should be able to get a reference implementation in C++ of Ladder instructions. So, instead of using the current black-box style of validation, we could consider some kind of white-box validation, by formally proving that the WhyML modelling simulates the C++ code. This however would require a significant effort, since it is somehow the same level of difficulty as proving the soundness of a compiler. See for example a work on a notion of ghost monitors using Why3 [7]. There should be simpler approaches though: we could double check the WhyML model versus the C++ code by generating and executing tests on both sides and compare the results. This could be done in two steps: generating tests from WhyML code and running the same tests in C++, and generating tests from C++ code and running the same test using the Why3 interpreter.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *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] Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. Explaining counterexamples with giant-step assertion checking. In José Creissac Campos and Andrei Paskevich, editors, *6th*

Workshop on Formal Integrated Development Environments (F-IDE 2021), Electronic Proceedings in Theoretical Computer Science, May 2021. URL: https://hal.inria.fr/hal-03217393, doi:10.4204/EPTCS.338.10.

[3] Cláudio Belo Lourenço, Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, and Hiroaki Inoue. Automated formal analysis of temporal properties of Ladder programs. *International Journal on Software Tools for Technology Transfer*, 24(6):977–997, 2022. URL: https://hal.inria.fr/hal-03737869, doi:10.1007/s10009-022-00680-0.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310. URL: http://hal.inria.fr/hal-00790310.

[5] 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. See also http://toccata.lri.fr/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[6] W. Bolton. *Programmable Logic Controllers (Sixth Edition)*. Newnes, 2015. doi:10.1016/C2014-0-03884-1.

[7] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive verification with ghost monitors. In *Principles of Programming Languages*, New Orleans, United States, 2020. URL: https://hal.inria.fr/hal-02368284, doi:10.1145/3371070.

[8] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL: https://hal.inria.fr/hal-01960203.

[9] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[10] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verication of Rust programs. In *International Conference on Formal Engineering Methods - ICFEM*, Lecture Notes in Computer Science, Madrid, Spain, 2022. Springer. URL: https://hal.inria.fr/hal-03737878.

[11] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer. URL: https://hal.inria.fr/inria-00270820v1, doi:10.1007/978-3-540-73368-3_21.

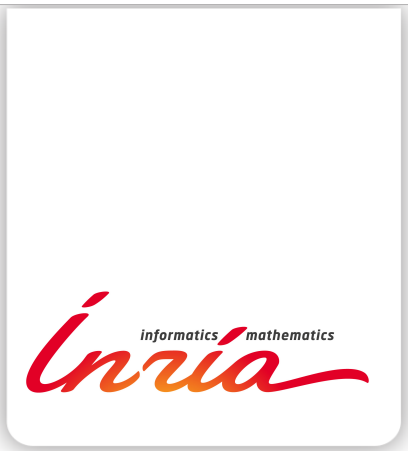
[12] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL: http://hal.inria.fr/hal-00789533.

[13] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes*

in Computer Science, pages 122–142, Rhodes, Greece, October 2020. Springer. See also http://why3.lri.fr/isola-2020/. URL: https://hal.inria.fr/hal-02696246.

[14] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016. Springer. URL: https://hal.inria.fr/hal-01344110, doi:10.1007/978-3-319-47166-2_32.

[15] 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.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803