Automated Deductive Verification for Ladder Programming

Denis Cousineau David Mentré
Mitsubishi Electric R&D Centre Europe (MERCE)
Rennes, France
{d.cousineau,d.mentre}@fr.merce.mee.com

Hiroaki Inoue
Mitsubishi Electric Corporation
Amagasaki, Japan
Inoue.Hiroaki@ah.MitsubishiElectric.co.jp

Ladder Logic is a programming language standardized in IEC 61131-3 and widely used for programming industrial Programmable Logic Controllers (PLC). A PLC program consists of inputs (whose values are given at runtime by factory sensors), outputs (whose values are given at runtime to factory actuators), and the logical expressions computing output values from input values. Due to the graphical form of Ladder programs, and the amount of inputs and outputs in typical industrial programs, debugging such programs is time-consuming and error-prone. We present, in this paper, a Why3-based tool research prototype we have implemented for automating the use of deductive verification in order to provide an easy-to-use and robust debugging tool for Ladder programmers.

1 Introduction

Programmable logic controllers (PLC) are industrial digital computers used as automation controllers of 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. Ladder language, also known as Ladder Logic, is a programming language used to develop PLC software. This language uses circuits diagrams of relay logic hardware to represent a PLC program by a graphical diagram. This language was the first available to program PLCs. It is now standardized in IEC 61131-3 [1] standard among other languages but is still widely used and very popular among technicians and electrical engineers.

In conventional development of software, a great part of the development time is dedicated to debugging. Debugging programs is crucial in the case of Factory Automation (FA) since bugs in factories can be extremely expensive in terms of human and material damages, and plant downtime. Debugging a Ladder program is particularly difficult, time consuming and costly. Bugs can be depicted as the violation, at some point of a program, of some property concerning values of inputs/outputs and local memory of the program. The objective of debugging consists in detecting those property violations before running the code in production, i.e. finding initial values of inputs and internal memory that lead to a property violation, when executing the program. Since it is almost impossible (and way too costly) to check all possible executions of a program, the usual method consists in developing and running some tests (i.e. executing the program on a particular initial configuration and check its behavior). In industry, tests used to be run directly in the factory, which is very costly and risky. Nowadays, most of the tests are run on a software simulation, but some are often still run in the factory for a last check of the program behavior in real conditions of use, or for bypassing the difficulty to simulate particular sequences of inputs. Even when run on a software simulation, tests-based processes are still time-consuming and cannot be exhaustive.

On the other hand, some research work has been done concerning formal analysis of Ladder programs. Most of this work [7] [6] [9] [10] concerns the verification of temporal properties of Ladder

R. Monahan, V. Prevosto, J. Proença (Eds.): Formal
Integrated Development Environment 2019 (F-IDE 2019)
EPTCS 310, 2019, pp. 7–12, doi:10.4204/EPTCS.310.2

© D. Cousineau, D. Mentré & H. Inoue

programs (a Ladder program being continuously executed in the PLC), and uses different model checking techniques. Some other work used deductive verification to detect data races [13] and prove safety properties [12] (with some temporal aspects) of Ladder programs. Model-checking techniques are limited by the state explosion problem they face when addressing real-world problems. On the contrary, deductive verification may give full confidence in the obtained results but may also prevent from a full automatization of the process (in terms of proof automation and specification formalization).

Our objective in this work was to make a proof of concept of an easy-to-use and robust tool for debugging Ladder programs, both increasing the quality of the code and decreasing the time to deploy, a crucial point in the context of Industry 4.0, in which assembly lines are more often reconfigured, hence code evolves frequently. We had to determine a good tradeoff to offer a high level of automation, together with providing a strong confidence in the given results (in particular when no bug is found). The solution we chose is similar to what SPARK/GnatPro [8] has done for Ada code. For the easy-to-use part, we targeted a full automated tool, so that it could be used by regular engineers with no needed knowledge in formal methods. We also focused on the information given to the programmer when an error is found, for easing errors fixing. In order to obtain a fully automated and complete tool, we targeted to detect runtime errors by the mean of deductive verification. We focused on runtime errors like integers overflows, divisions by zero, violations of Ladder instructions' pre-conditions, etc...

We based our prototype implementation on the Why3 platform [4]. Why3 offers an expressive formalization language, an efficient Weakest-Precondition (WP) calculus [5] implementation and a rich API to send the obtained verification conditions to several automated solvers. Moreover, with its *labels* mechanism, Why3 allowed us to keep code information during the whole automatic process, for providing rich and useful information to the programmer in case a bug is detected.

2 Ladder Logic

Ladder Logic is a graphical programming language using relay logics diagrams to represent a PLC program. A Ladder program takes inputs values (*contacts*) that correspond to the fact that physical relays are wired, not wired, pulsing (rising edge) or downing (falling edge) and other values stored in the internal memory of the PLC (booleans, 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 contacts values). Coils and instructions are activated when the combination of contacts at their left gives a *wired* value, and they can also be parallelized (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 (a *diagram*) is composed of several rungs.

Figure 1: Ladder program example

Figure 1 depicts a very simple Ladder example. This program has one contact `X0`, and when the physical

relay corresponding to that input is activated, the program calls instruction INC that increments the value of its device argument D0, and then calls instruction BCD with D0 and D1 devices as respectively input and output arguments. The BCD instruction converts a 16 bits integer into a 16 bits BCD (Binary-Coded Decimal) integer. The 16 bits BCD format represents 4 digits decimal numbers, using 4 bits to represent each of the 4 digits. It is typically used for display purpose. Since this format can only represent 4 digits decimal numbers, the *BCD* instruction raises an error when it is called on a device value that does not belong to interval $[0;9999]$. This is typically the kind of runtime errors we want to detect with the tool we developed. Regarding this example, we were also interested in overflows that could occur when calling instruction INC. The example we present is very simple but typical industrial programs we had access to have hundreds of lines, hundreds of inputs, devices and outputs, and dozens of instructions calls. As a last point, 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*.

3 Modelling Ladder in Why3

We chose not to model the temporal/cyclic aspect of execution of Ladder programs, but only one scan in order to detect error scenarios, i.e. values of inputs and devices at scan beginning (before execution) that may lead to a runtime error. We developed a library of Ladder instructions formalizations. We depict here the formalization of the BCD instruction. This formalization is composed of two functions. The first one, `bcd_compute`, computes the 4 digits of an decimal integer argument and returns the decimal value of the BCD representation of those 4 digits. The second one, `bcd` takes three arguments: `input` is the wiring value of the line to which the instruction is connected, `src` is the value of the input device, and `prev_val` is the value of the output device before execution of the instruction. The `requires` pre-condition states that either `input` does not activate the instruction or `src` must belong to interval $[0;9999]$. You can notice the two strings *labels* in the pre-condition. The first allows asking solvers to find a counter-example if they cannot prove the verification conditions associated with that pre-condition. The second one allows keeping semantic information during the whole process, in order to give back this information to the programmer in case an error scenario is found. The `returns` post-condition states that the function returns the previous value of the output device when the instruction is not activated, and the actual BCD computation otherwise.

```
(***** BCD *)
function bcd_compute (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  (input : pulse)
         (src : int)
         (prev_val : int) : int
requires { "model_vc"
           "expl:BCD: out of [0...9999] range call"
         (inactive input \/ (0 <= src <= 9999))
        }
returns {result -> active input -> result = bcd_compute src }
returns {result -> inactive input -> result = prev_val }
=
if active input then
 bcd_compute src
else
 prev_val
```

Figure 2: Why3 formalization of Ladder BCD instruction

We developed around fifty such formalizations of Ladder instructions in order to run our tool prototype on the industrial program samples we had access to. Then our translation of Ladder programs to Why3 models consists in translating on-the-fly the logical expressions that correspond to coils and their combinations, and combine them with calls to the instructions formalizations of our library, using a single-state-assignment [11] transformation to handle the iterative aspect of Ladder programs.

4 Prototype architecture

Our tool automatically ① translates Ladder programs into Why3 modules that refer to the instructions formalizations described in the previous section. We implemented our own library to produce Why3 text files, to help the reuse of generated modules. During the translation, we use *labels* to keep information on code location of instruction calls to give improved feedback to the programmer when an error is found.
Then we use Why3's WP calculus to ② compute verification conditions that correspond to preconditions of instructions, and use Why3 API to ③ send those verification conditions to SMT-solver CVC4 [3] (we chose CVC4 for its overall good performances and its ability to generate counter-examples when a verification condition cannot be proved). Couter-examples are then ④ interpreted as initial values of the original program and simulated execution ⑤ recomputes, from those inital values, all the intermediate values of devices, wires, etc... from the beginning of the program to the location where the error occurs. Finally, we ⑥ provide a graphical feedback to the programmer, with those intermediate values information and informations concerning the error the tool found.

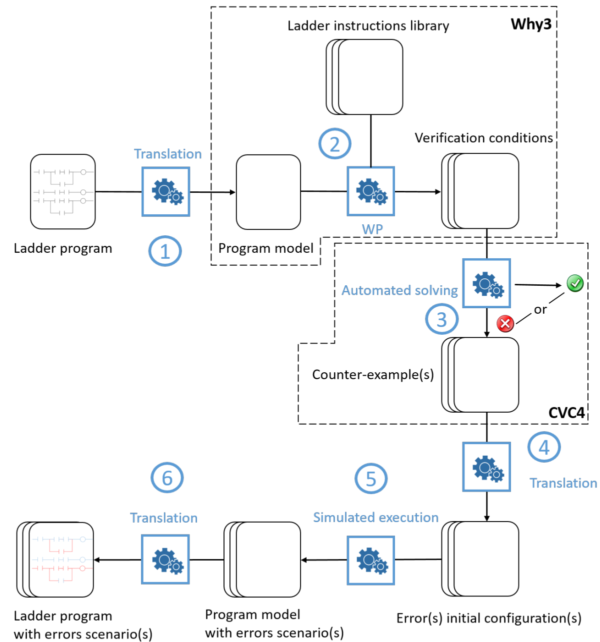

Figure 3: Prototype architecture

5 Graphical user feedback

We implemented a proof of concept of a graphical interface which gives back to the programmer information about found bugs, in an easy-to-understand manner. The aim of such an interface is to be directly integrated in Ladder IDEs. We based our prototype implementation on the Ocsigen web framework [2] which allowed us to quickly prototype a web-based graphical interface displaying information coming from our tool prototype implemented in OCaml. We identified three pieces of information that should be displayed to the programmer when a bug is found: the *error location* (where the error occurs), the *error reason* (why the error occurs) and the *error scenario* (when the error occurs). The error location is encoded during the on-the-fly translation from the Ladder program to the Why3 model: to each instruction call is attached a label which contains its location in the original source code. This label is propagated during the WP calculus, appears in the verification condition sent to the automated solver and comes back in the counter-example the solver gives when it finds one. The error reason is encoded in the Why3

instructions library as shown in figure 2. It is attached, with a "expl:" label, to pre-conditions of Ladder instructions, and is propagated during the whole process, like code locations labels. As explained in the previous section, the error scenario consists in the initial and intermediate values that lead to the error. It is re-computed from solvers' counter-examples, and is expressed with colors for wiring values (blue when a wire is active, grey otherwise) and figures for other values above the corresponding devices.

Figure 4 shows a screenshot of this graphical interface. This is what our prototype returns when run on example of figure 1. In this case, the interface states the errors occurs at BCD instruction call location (it is colored in red). It also states that the error reason is an out-of-range call. And it gives the error scenario: contact X1 is active hence colored in blue, then the wires at its right are also active and colored in blue. After execution of instruction INC, value stored in device D1 is 10,000, which leads to the error, when given as argument of instruction BCD.

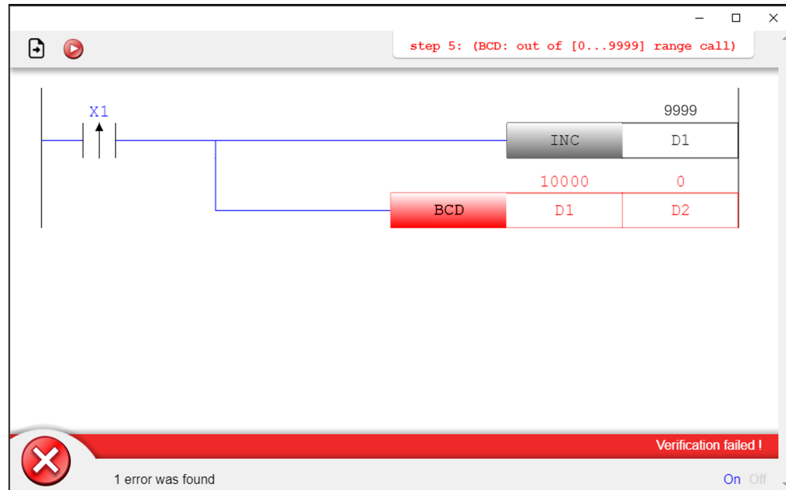

Figure 4: Graphical feedback

We believe that this interface, somehow inspired by what already exists in Ladder simulation software, in particular for the colors, may be very useful for debugging Ladder programs, in particular when they reach a critical size with hundreds of rungs, inputs, devices, outputs, etc... Indeed, it is much easier to understand why an error occurs with this kind of interface, than when using tests, in which case only the initial configuration of the program is given.

6 Performances

Our proprietary prototype is implemented in OCaml, in about 13,000 lines of code, including 3,000 lines for our library to produce Why3 textual files and 4,000 lines for the graphical user interface. We made some optimization effort concerning the modelization of Ladder language in Why3, in order to obtain a fully automated and fast process. But we made no optimization in our OCaml code, and even did not parallelize the calls to SMT-solvers. Nonetheless our protype has already pretty good performances. We ran our prototype tool on an industrial code sample with 1,657 *steps* (i.e. contacts, coils and instruction calls), among which three instructions calls could lead to an error. On a virtualized Ubuntu 18.04, running in VirtualBox 5.2, under Windows 10 on a Intel Core i7-7500U 2.70 GHz laptop, it takes only 3 to 4 seconds for our prototype to answer. Almost all the time is taken by CVC4 (about one second for each of the three verification conditions that are handled sequentially). This gives us confidence in the fact that the technology and architecture we chose are relevant for the implementation of a real industrial tool.

7 Conclusion

The objective of this work was to make a proof of concept of a formal methods-based debugging tool for industrial Ladder programs. For such a debugging tool to be incorporated in an industrial process, we

think that it should be transparent and bring strong added value to the user. First, our proof of concept shows that such a formal debugging tool can be transparent: it needs no specific knowledge since all the process is fully automatic (Ladder programmers do not need to write a formal specification, and even less a model of their codes); it is very fast so it may be run during the programming phase of the development process and not in a separated phase; it may be fully integrated in a Ladder IDE as our GUI prototype shows. Second, our proof of concept shows the added value such a tool could have in regard to current debugging tools: our prototype can give back to the programmer very precise and useful information when it detects an error (using a intelligible interface); and the deductive verification technique we used, thanks to the Why3 platform, gives a strong confidence when the tool detects no runtime error (since it is equivalent to test all possible inputs and devices values configurations).

A drawback of our prototype concerns the fact that it may raise false positive alarms, since it only considers one scan of the Ladder program. For example in Figure 1, value of device `D0` may be changed *after* the `BCD` instruction call, such that value $10,000$ is never reached. Nevertheless, our prototype would still raise an alarm. In future work, we plan to decrease the number of false alarms by considering a few consecutive scans in our Why3 Ladder formalization.

Another way to improve our prototype could be to provide some *quickfix*-like mechanisms to programmers. In example of Figure 1, our prototype could propose to the programmer to add automatically, before the `BCD` instruction call, a line that resets `D0` when it does not belong to range $[0;9999]$.

References

[1] (2013): *IEC 61131-3:2013, Programmable controllers - Part 3: Programming languages*.

[2] V. Balat (2006): *Ocsigen: Typing Web Interaction with Objective Caml*. In: *ACM SIGPLAN workshop on ML*, Portland, United States, doi:10.1145/1159876.1159889.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds & C. Tinelli (2011): *CVC4*. In: *Computer Aided Verification*, CAV'11, Springer-Verlag, doi:10.1007/3-540-45657-0_40.

[4] F. Bobot, J-C. Filliâtre, C. Marché & A. Paskevich (2015): *Let's Verify This with Why3*. *Software Tools for Technology Transfer (STTT)* 17(6), pp. 709–727, doi:10.1007/s10009-014-0314-5.

[5] E. W. Dijkstra (1997): *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[6] B. Fernández Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech & V. M. González Suárez (2015): *Applying Model Checking to Industrial-Sized PLC Programs*. *IEEE Transactions on Industrial Informatics* 11(6), pp. 1400–1410, doi:10.1109/TII.2015.2489184.

[7] G. Frey & L. Litz (2000): *Formal methods in PLC programming*. In: *IEE International conference on systems, man and cybernetics*, 4, pp. 2431–2436 vol.4, doi:10.1109/ICSMC.2000.884356.

[8] D. Hauzar, C. Marché & Y. Moy (2016): *Counterexamples from Proof Failures in SPARK*. In: *Software Engineering and Formal Methods*, Springer, doi:10.1007/978-3-662-49674-9_25.

[9] S. Kottler, M. Khayamy, S. R. Hasan & O. Elkeelany (2017): *Formal verification of ladder logic programs using NuSMV*. In: *SoutheastCon 2017*, pp. 1–5, doi:10.1109/SECON.2017.7925390.

[10] T. Ovatman, A. Aral, D. Polat & A. Osman Ünver (2014): *An overview of model checking practices on verification of PLC software*. *Software and Systems Modeling*, pp. 1–24, doi:10.1007/s10270-014-0448-7.

[11] B. K. Rosen, M. N. Wegman & F. K. Zadeck (1988): *Global Value Numbers and Redundant Computations*. In: *Symposium on Principles of Programming Languages*, POPL '88, ACM, pp. 12–27, doi:10.1145/73560.73562.

[12] J-M. Roussel & B. Denis (2002): *Safety properties verification of ladder diagram programs*. *Journal Européen des Systèmes Automatisés (JESA)* 36(7), pp. pp. 905–917.

[13] Z. Su (1997): *Automatic Analysis of Relay Ladder Logic Programs*. Technical Report UCB/CSD-97-969, EECS Department, University of California, Berkeley.