Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs

O. Rossi
University Laboratory in Automated Production Research
fax: 33 1 47 40 22 20
Ph. Schnoebelen
Laboratory for Specification et Verification
ENS de Cachan, 61 av. du président Wilson 94235 Cachan cedex
email: rossi@lurpa.ens-cachan.fr ; phs@lsv.ens-cachan.fr

Abstract

We describe our approach to the automated verification of Ladder Diagrams programs. This combines a formal semantics for a large fragment of the LD language (including a modeling of timed function blocks), and a powerful temporal logic model checking technology.

Keywords: formal verification, programmable logic controllers, real-time, Ladder Diagrams, IEC 61131-3.

1 Introduction

1.1 Contents of the paper

We describe a formal approach to the automated verification of Programmable Logic Controller (PLC) programs written in the Ladder Diagrams (LD) language of the IEC 61131-3 standard [4]. Our approach introduces a formal semantics for LD programs in terms of state automata, on which we use Cadence SMV [6], a powerful symbolic model checking tool, that checks automatically whether the LD program satisfies its expected behavioural properties. The properties are written in temporal logics (CTL and LTL).

Our formal semantics for LD programs accounts for a large subset of the full language. This includes rising and falling edges, jump instructions, and any intricate sequencing of rungs. Regarding real-time constructs, we deal with *Time ON delay* function blocks (TON) at a logical level only, allowing the proof of liveness and safety properties that require the eventual triggering of TONs.

Throughout the paper, the same non-trivial example is used to illustrate the method.

1.2 Related works

Several people already proposed formal behavioral models of PLC programs, with an eye towards using model checking for verification purposes. Timers are considered in [5], applied to the Instruction List language. Two different modelings using timed automata are presented, but no actual example of model checking is given to illustrate the feasibility of the approach.

Regarding LD, [1] models LD programs as constraints systems. Jumps and edges are not taken into consideration. The model is used for simulation, not for exhaustive model checking. [7] models LD programs as state automata and uses SMV for symbolic model checking. However, the extraction of a state automaton out of a LD program is not discussed. It is only described on a simple example and would not extend easily to more complex programs. [8, 11, 9] is another line of work towards the formal validation of PLC programs. There LD programs are modeled as state automata, while the plant is modeled with Condition/Event systems. Properties are then proved using model checking. Like [7], these works lack a general formal definition of how a given LD program gives rise to a state automaton. Also, timers, jumps, rising and falling edges are not taken into account.

2 Working example

We consider a functional subset of MIX2BRIX (see [4, annex F5]). The physical subset is a tipping mixer that mixes products from a previous system and pours them into the next system of the production plant (see Fig. 1). The inputs and outputs of the system are listed in Fig. 2.

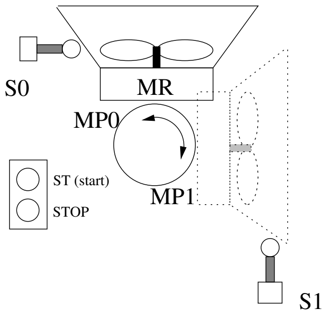

Figure 1: Tipping mixer physical model

2.1 Nominal behavior of the mixer

From an initial state (all actuators stopped, mixer upright), the *start* button is pushed (ST: $0 \to 1$) setting the mixing motor on (MR: $0 \to 1$). After a duration t_1 is elapsed, the bowl starts tipping (MP1: $0 \to 1$) until sensor S1 detects that the tipped position is reached (S1: $0 \to 1$). Then mixing and tipping stop (MR, MP1: $1 \to 0$) and the mixer moves back towards its original upright position (MP0: $0 \to 1$) until S0 signals completion. At any time, a STOP signal would override this nominal behavior: its effect is to stop all actuators no matter what the current state is. When the STOP signal disappears, the system regains its initial state, waiting for a new occurrence of ST.

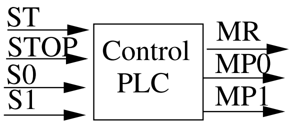

Figure 2: System inputs and outputs

For controlling this mixer, we designed the LD program given in Fig. 3.

```
   |    STOP                     ESTOP  |
1  +---| |---------------------->>      |
   |    STOP                     INIT   |
2  +---|N|-------+--------------->>     |
   |    INITOK   |                      |
   +---|/|-------+                      |
   |                                    |
   |    ST    MP1   MP0           MR    |
3  +---|P|---|/|---|/|----------(S)--+
   |                                    |
   |    ST     MP1   MP0   S1    MST    |
4  +---|P|-+--|/|---|/|---|/|--( )--+
   |    MST |                           |
   +---| |-+                            |
   |                +--------+          |
   |    MST         |        |    MP1   |
5  +---| |----------+  TON   +---(S)--+
   |                |        |          |
   |           t1---+        |          |
   |                +--------+          |
   |     S1   MP1                 MP1   |
6  +---| |---| |---------------+--(R)--+
   |                           |  MP0   |
   |                           +--(S)--+
   |                           |  MR    |
   |                           +--(R)--+
   |                                    |
   |     S0                       MP0   |
7  +---| |----------------------(R)--+
   |                                    |
8  +----------------------<RETURN>      |
   ESTOP                                |
   |                              MR    |
9  +---------------------------+--(R)--+
   |                           |  MP0   |
   |                           +--(R)--+
   |                           |  MP1   |
   |                           +--(R)--+
   |                           |INITOK  |
   |                           +--(R)--+
   |                           |  MST   |
   |                           +--( )--+
   |                                    |
10 +----------------------<RETURN>      |
   INIT                                 |
   |   S0                         MP0   |
11 +--|/|-----------------------(S)--+
   |                                    |
   |   S0                       INITOK  |
12 +--| |--------------------+--(S)--+
   |                           |  MP0   |
   |                           +--(R)--+
   |                                    |
```

Figure 3: Mixer LD control program

3 An operational semantics for LD programs

The operational semantics for LD is a formal definition, in terms of *state automata* (or *Kripke structures* in temporal logic speak), of the abstract behavior of a LD program. This section is only an overview and we refer to [10] for the full formal definition.

3.1 Working hypotheses

We started from the IEC 61131-3 standard but, since the language it describes is too complex (both syntactically and semantically) and is only described informally, we only consider a fragment of the LD language.

These restrictions aim at simplifying the definition of a formal semantics but the considered fragment LD matches the expressiveness and the complexity of industrial LD programs: all contacts described in the IEC 61131-3 are taken into account, plus TON function blocks, set/reset output coils and jump instructions.

More precisely, we restrict ourselves to programs where:

(R1) All variables are boolean (e.g., no integer counter function blocks),

(R2) Each LD rung is composed of a *testing part* (test contacts and combination wires) followed by an *assignment part* (output coils) [1].

(R3) There is only one LD program in the target PLC.

Furthermore some important aspects of LD program behavior, as described in the IEC 61131-3 standard, are not enforced in all implementations of Relay Ladder Logic languages. These are:

(R4) the rungs are evaluated *sequentially*, from top to bottom,

(R5) *evaluating* a rung means first computing the result (a boolean value) of the testing part of the rung, and then using this value to update the variables in the assignment part (according to [4], the output of a coil cannot be updated unless its input has been computed),

(R6) this strict sequencing can be interrupted by *jump instructions* from the assignment part.

Finally, we assume the LD program runs in a *cyclic PLC*. Though such PLCs are widely used in the industry, their behavior is not described in the IEC 61131-3 standard. They repeat the following *execution cycle*: first read the values of the input variables from the physical environment, then latch them in the memory, then run the complete LD program once (with input variables remaining constant), then latch the values of the output variables to the environment and start a new cycle.

3.2 Configurations and transitions for LD programs

With a given LD program is associated its set of *configurations*. A configuration is an instantaneous descrip-

[1] This restriction follows the industrial guidelines for LD programs design and is not a limitation *per se*.

tion of the program state (a snapshot) and is formally defined as a valuation of all variables of the program (i.e., a mapping from variables to values). These variables include all declared variables (input, internal, output and other variables) and a special "current control location" variable, called `lc`, that ranges over all points of the execution cycle.

Then the operational semantics formally describes, by means of a relation Φ between configurations, how a configuration c is followed by a configuration c'. The definition of Φ has two main cases:

1. *Within the execution cycle*, the system moves from a configuration c to the next one c' by evaluating the current (in c) rung. Here c' is uniquely determined by c. In configuration c, the testing part of the rung yields the boolean result v. Then c' is obtained from c by updating with v the coils in the assignment part of the rung. The control variable, `lc`, is updated according to jump instructions if any are present in the assignment part (otherwise `lc` is simply incremented). All other variables remain unchanged.

2. *At the end of the execution cycle*, which corresponds to the maximum value for `lc`, the system reads from the environment the new values of its input variables and sets `lc` to 0 for a new cycle. The model allows arbitrary evolution of the inputs so an end-of-cycle configuration is linked to 2^n successor configurations (corresponding to all possible values for n input variables).

Formally, Φ is described via a mathematical formula which is easily written in the input language of a symbolic model-checker (as done in [6]). Some important fragments of that definition for Φ (e.g. the behavior of the `lc` control variable) appear in section 5 below.

We developed a translator that accepts LD programs and produces a definition of the associated state automaton (in SMV's input language). For parsing reasons, the translator starts from a textual equivalent of graphical LD programs. This textual form profits from restriction (R2) but still allows rungs, contacts, coils, combination elements and functions blocks. The prototype translator we use is currently evolving towards accepting LD programs in a syntax closer to the graphical form.

4 Handling Time ON delay (TON) function blocks

Time ON delay function blocks (TON) are primitives for describing timing delays in LD programs. We considered them in our study because they are widely used in industrial applications, and their behavior is fully described in the IEC 61131-3 standard. Fig. 4 describes the inputs and outputs of a TON block. A TON block may appear in the testing part of a rung (see, e.g., rung 5 of Fig. 3).

Figure 4: Inputs and outputs for TON blocks

The standard definition of the behavior of a TON block is given via a logic chronogram [4]. An internal float variable `et` measures the *elapsed time* (increases along with the internal clock of the PLC) as long as input a holds true, and then output `Q` is false. `et` is set to 0 when a is false. `et` is also compared to input t_1: if $\texttt{et} = t_1$, then `Q` is set to true, and `et` is no longer increased. Our formal model is an abstract simplification of the above behavior. The explicit timing values are not taken into account, and we only keep a logical view of the trigger condition ($\texttt{et} = t_1$).

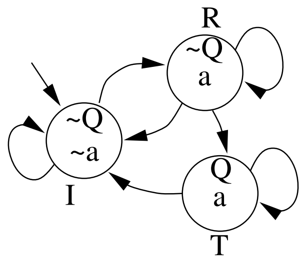

Figure 5: TON behavior abstraction

The abstract model of a TON block has 3 states: inactive (`I`), running (`R`) or triggered (`T`). The transitions are shown in Fig. 5: a running TON can either stay in state `R` or trigger, in a nondeterministic way. The output `Q` is set to true when the TON is triggered. If, during some evolution of the TON block, the input `a` holds false, then the state of the TON is set to `I`.

This finite-state logical view of the trigger condition of a TON block leads to a simple model that can be smoothly integrated in the transition relation for the rest of the system.

Compared to more quantitative modelings of the real-time aspects (e.g. using timed automata *à la* Alur and Dill [2]) our abstracted view leads to tremendous computational simplifications. Our approach has its limitations (the triggering order of two distinct blocks can not be discriminated, and explicit time measurement is not allowed) but it is safe: any safety property satisfied by our model is also true in the quantitative model based on timed-automata. The converse is not true but, in practice, there are relatively few examples of interesting properties that only hold in the finer model.

4.1 Integration into the operational semantics

When TON blocks are used, the operational semantics from section 3 must be modified. Assume a TON block appears in rung i: an additional internal variable encodes the internal state (`Q`, `R` or `I`) of that TON block. Evaluating rung i now requires two steps. First the system moves from c to c' by evaluating the next state of the TON block, which may depend on the values of inputs in c. Here c' is not uniquely determined from c since `et` may become true or false non-deterministically. Then the current value of output `Q` is known and the usual evaluation of a rung can be resumed. This behavior is modeled in the associated state automaton by inserting a new control point in each rung where a TON block occurs. The chosen way for encoding this is to associate a boolean variable `fbe` (*function block evaluation*) to the TON block, and to move control from "`(lc=i)&(fbe=false)`" to "`(lc=i)&(fbe=true)`" to "`lc=i'`"...

When there is more than one TON block in a rung, they are sequentially evaluated in the natural left-to-right, top-to-bottom order. The evaluation of a rung with n TON requires $n+1$ steps.

5 CADENCE SMV IMPLEMENTATION

The input language of Cadence SMV allows writing down the definition of Φ, the transition relation. The chosen approach is modular: for each variable (program variable or control variable), we write constraints between before and after values (possibly depending on the other variables).

For each rung without TON, the boolean result v (cf. section 3) generated by the *testing part* is defined as an alias. It is a function of all the variables in the *testing part*. For example, the *testing part* of the second rung of the Fig. 3 is declared as follows:

```
#define rung4
  (RE__ST | MST) & ~MP1 & ~MP0 & ~S1
```

`RE__ST` stands for the value of the rising edge contact associated with the `ST` variable. `MST` stands for the value of the normal contact associated with the `MST` variable. The computation of `RE__ST` is defined below. The LD language grammar defines the logical combination between the values associated with the variables of the *testing part*. | (respectively `&`, `~`) stands for the logical operator OR (respectively AND, NOT).

The rising and falling edges are also defined as aliases. These language primitives detect a change in the value of the associated variable between two execution cycles. For each variable associated with an *edge* contact, a memorization variable is created. It stores the value the original variable held at the end of the previous cycle (`eoc : lc=12`). `RE__ST` and `MEMO__ST` are declared as follows:

```
#define RE__ST (~MEMO__ST & ST)
next(MEMO_ST) := case
   {
    eoc : ST;
    default : MEMO_ST;
   };
init(MEMO_ST) := ST;
```

The definition of `MEMO__ST` refers to `eoc` (end of cycle), a specific value of `lc`, the control variable (`lc`=12). The definition of `lc` enables to execute rungs one after the other, in the declaration order. Jump instructions, as in the first and second rung of the program in Fig. 3 alter this implicit ordering. If a jump instruction is executed, the next rung to be evaluated is the rung placed below the jump instruction target. They are taken into account in the definition of `lc`:

```
next(lc) := case
  {
    lc=0 & rung1 : 8;
    lc=1 & rung2 : 10;
    lc=7 & rung8 : 12;
    lc=9 & rung10 : 12;
    lc=4 & rung5 & ~fbe : lc;
    default : lc+1 mod(13);
  };
init(lc) := 0;
```

The encoding of the TON abstraction is done using the same declarative method. The state of each TON is represented using a 3-state variable: `t_state`. The evolution of the TON state depends upon the program control variable (`lc`), the TON control variable (`fbe`), the current state value (`t_state`), and the input variables. In our example, the value of the output variable is given by rung 5, which is the testing part placed before the TON. The corresponding code is given below. As shown in Fig. 5, 'i' stands for 'inactive', 'r' stands for 'running', and 't' stands for 'triggered'.

```
next(t_state) := case
{
  lc=4 & ~fbe & ~rung5 : i;
  lc=4 & ~fbe & rung5 & t_state=i : r;
  lc=4 & ~fbe & rung5 & t_state=r : {r,t};
  lc=4 & ~fbe & rung5 & t_state=t : t;
  default : t_state;
};
init(t_state1) := i;
```

When the TON is running and its input variable is still true, the next state can either be 'running' or 'triggered', non-deterministically. The last ingredient is a fairness constraint that we postpone until section 6.3.

Now the state automaton model has been defined and SMV expects an assertion section, listing temporal logical properties to be checked.

6 AUTOMATED PROOF OF TEMPORAL PROPERTIES

Cadence SMV is able to analyze the state automaton and automatically check behavioral properties expressed in temporal logics. These temporal logics are tailored for linking individual instantaneous properties in complex temporal (i.e., "before-and-after") ways, where boolean combinations and arbitrary nesting of temporal modalities are allowed.

Here, we assume the reader is familiar enough with CTL and LTL (see e.g. [3]), the temporal logics used in Cadence SMV, and we only write temporal formulae and their reading in plain English.

6.1 Invariance and reachability

Verifying the LD program from Fig. 3 starts with simple invariance and reachability properties.

```
safe_motor: assert G ~(MP0 & MP1);
```

`G (MP0 & MP1)` states that, in all configurations along all runs of the system, `(MP0 & MP1)` holds, i.e. the two outputs `MP0` and `MP1` are not both on. Hence `safe_motor` states that the tipping motor is never given contradictory orders.

```
safe_mix1: assert G (MP0 -> MR);
safe_mix2: assert G (MP1 -> ~MR);
```

Using the same pattern, `safe_mix1` (resp. `safe_mix2`) states that in all configurations tipping the bowl down ("`MP0`") is done while mixing ("`MR`"), and tipping up ("`MP1`") while not mixing.

```
for (i = 0; i <= 12; i = i+1)
   reachable[i]: SPEC AG EF (lc=i);
```

For a given i, `reachable[`i`]` states that, whatever happens in the system ("AG"), rung i (or end of cycle, for $i = 12$) remains reachable ("`EF (lc=i)`").

6.2 Liveness properties

```
end_1 : assert G (S1 -> F ~MP1);
end_0 : assert G (S0 -> F ~MP0);
```

`end_1` states that, along all runs ("`G`") if `S1` holds then somewhere later along the run ("`F`") `MP1` will hold, that is, whenever the down sensor reacts, the tipping motor will eventually be stopped. `end_0` states a similar property for the other tipping motor.

Similarly, we state that whenever the start button is pushed ("`ST`"), the system will eventually ("`F`") start tipping ("`MP1`") unless a stop signal is sent in the meantime ("`| STOP`").

```
live : assert G (ST -> F (MP1 | STOP));
```

6.3 Hypothesis for TON behavior

That property `live` holds crucially depends on the fact that, after a duration t_1, the TON block trigger. However, the operational model of timers we described only *allows* eventual triggering. It does not enforce it. Hence Cadence SMV rightfully states that property `live` is not satisfied.

We can restrict the non-deterministic behavior in a minimal way and ensure that the TON *will eventually trigger* by associating with the definition of Φ a fairness constraint (a temporal logic formula restricting the set of possible runs) stating that if the TON is in the 'r' state and infinitely often evaluated, then, if the input variable holds, the TON will eventually trigger.

Strictly speaking, this fairness constraint is part of the model and should have appeared in section 5. However, with SMV it is more natural to see this as a behavioral assumptions that is used for checking some temporal formulae. One benefit of this view is that it let us see when the fairness constraint is required (e.g., for property `live` but not for `safe_mix1`).

The assumption is:

```
finite_time_count: assert ((F G (~rung5
        | ~(lc=4)|fbe))| (G F (rung5
        & lc=4 & ~fbe & ~(t_state=e))));
```

which reads "*either the system ends up never using the TON block, or the TON transition from* `R` *to* `T` *is always eventually used*".

With this assumption, Cadence SMV is able to check that `live` holds when we issue the directive:

```
using finite_time_count prove live;
```

6.4 Checking the observable behavior

One feature of the operational model described in section 3 is its fine-grained view: the execution cycle is divided into several transitions, and even the evaluation of a single rung may lead to several consecutive transitions. This allows accounting for complex control structures. The temporal logic formulae checked by Cadence SMV refer to this view: e.g., checking that `AG (~MP0 & ~MP1)` proves that the two variables are never true at the same time *even within the evaluation cycle*.

In some cases, this viewpoint is not adequate (e.g., when a safety invariant seem to be violated during transient configurations). Property `end_1` above is another example: in reality, one wants that `~MP1` eventually occurs in an observable state (i.e., at the end of an execution cycle, `lc=eoc`, according to the execution pattern of the PLC).

An original approach to this problem has been developed within the framework of our study: there exists an automatic way of encoding temporal properties expressed at the observable levels into properties expressed at the fine-grain level. This way, the same formal model and the same model-checker can be easily used for the two levels.

7 Conclusion

We presented a formal approch to the automatic verification of Ladder Diagram programs, written according to the IEC 61131-3 standard. This combines a formal semantics, in terms of state automata, of LD programs, and symbolic model checking of temporal logic properties.

One of the most important aspects of our method is that the formal model deals with the temporal or logical, not the real-time or quantitative, view of the timing primitives. This avoids the huge complexity of timed-automata while still allowing to automatically prove many liveness and safety properties in programs that include timed function blocks.

The work we described is still in progress. We aim at extending LD language definition and semantics, in order to match the graphical form of LD diagrams and to include other standardized temporal primitives such as Time Off-delay (TOF) and Time Pulse (TP) function blocks.

REFERENCES

[1] A. Aiken, M. Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. In *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal, March 1998*, volume 1384 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 1998.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[4] IEC (International Electrotechnical Commission). *IEC Standard 61131-3 : Programmable controllers - Part 3*, 1993.

[5] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proc. 11th Euromicro Conference on Real Time Systems, York, England, June 1999*, pages 114–122. IEEE Comp. Soc. Press, 1999.

[6] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[7] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems*, 14(2):53–59, 1994.

[8] S. T. Probst. *Chemical Process Safety and Operability Analysis Using Symbolic Model-Checking*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, USA, May 1996.

[9] M. Rausch and B. H. Krogh. Formal verification of PLC programs. In *Proc. American Control Conference, Philadelphia, PA, USA, June 1998*, pages 234–238, 1998.

[10] O. Rossi, O. De Smet, and G. Canet. Définition et sémantique opérationnelle du langage LD0. Manuscript available from the first author, March 1999.

[11] A. L. Turk, S. T. Probst, and G. J. Powers. Verification of real time chemical processing systems. In *Proc. Int. Workshop Hybrid and Real-Time Systems (HART'97), Grenoble, France, Mar. 1997*, volume 1201 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 1997.