LTL-specification, verification and construction of PLC programs

Ryabukhin D. A.
Yaroslavl State University
Yaroslavl, Russia
Email: dmitriy_ryabukhin@mail.ru

Kuzmin E. V.
Yaroslavl State University
Yaroslavl, Russia
Email: kuzmin@uniyar.ac.ru

***Abstract*—An approach to specification, verification and construction of PLC programs for discrete problems is proposed. For the specification of the program behavior, we use the linear-time temporal logic LTL. Programming is carried out in ST, IL and LD languages according to an LTL-specification. The correctness analysis of an LTL-specification is carried out by the symbolic model checking tool Cadence SMV. A new approach to PLC-programming is shown by an example. For a discrete problem, we give an ST-program, its LTL-specification and an SMV-model.**

I. Introduction

Application of programmable logic controllers (PLCs) for systems controling complex industrial processes makes exacting correctness demands to PLC-programs. Any software error is considered to be inadmissible. However, the existing development tools for programming PLC, for example widely known CoDeSys (Controller Development System) [8], provide only usual debugging facilities through testing programs (not guaranteeing total absence of errors) by means of a visualization of PLC-control objects. At the same time certain theoretical knowledge and experience of applying the existing developments in the field of formal methods of modeling and analysis of software systems are accumulated. The programming of logical controllers is a practical area, in which existing developments could have successful application. Successful application is understood as implementation of formal methods in the process of program design at the level of a well-functioning technology which is clear to all specialists involved in this process — engineers, programmers and testers. Being as usual of a small size and having a finite state space, PLC-programs are exceptionally convenient objects for the formal (including automatic) analysis of correctness.

Programmable Logic Controllers (PLCs) are a special type of a computer widely used in automation systems [10], [5]. A PLC is a reprogrammable computer, based on sensors and actors, which is controlled by a user program. They are highly configurable and thus are applied to various industrial sectors. A PLC is a reactive system. A PLC repeats the execution of a user program periodically. There are three main phases for program execution (working cycle): 1) reading from inputs (sensors) and latching them in the memory, 2) program execution (with input variables remaining constant), 3) latching the values of the output variables to the environment.

Programming languages for logic controllers are defined by the IEC 61131-3 standard. This standard includes the description of five languages: SFC, IL, ST, LD and FBD.

IL (Instruction List) is an assembly language with an accumulator and jumps to labels. IL allows to work with any data types, to call functions and function blocks, written in IEC 61131-3 standard languages. IL is used to build small components, when critical control is required. Instructions are executed with the accumulator content. The IL accumulator is a universal container, which can keep values of any type.

ST (Structured Text) is high-level programming language. Its syntax is the adapted Pascal syntax.

LD (Ladder Diagram) represents a program by a graphical diagram based on circuit diagrams of relay logic hardware. The language itself can be seen as a set of connections between logical checkers (contacts) and actuators (coils). If a path can be traced between the left side of the connection and the output, through asserted contacts, the output coil storage bit is asserted true. If no path can be traced, the output is false.

This languages provides a possibility of application of all existing methods of program correctness analysis — testing, theorem proving [9] and model checking [7] — for verification of PLC-programs. Theorem proving is more applicable to “continuous” stability and regulation problems of the engineering control theory, since an implementation of these problems in PLC is associated with programming of an appropriate system of formulas. Model checking is most suitable for “discrete” problems of logical control, requiring PLCs with binary inputs and outputs. This provides a finite space of possible states of PLC-programs.

The most convenient for programming, specification and verification of PLC-programs are ST, LD and SFC languages, since they do not cause difficulties for neither developers nor engineers and can be easily translated into languages of software tools of automatic verification.

Earlier in the article [2], a review of methods and approaches to programming “discrete” PLC problems was carried out on languages LD, SFC and ST. For these approaches the usability of the model checking method for the analysis of program correctness with respect to the automatic verification tool Cadence SMV [13] was evaluated. Some possible PLC-program vulnerabilities arising at traditional approaches to programming of PLC was revealed. In particular, existing

This work is licensed under the Creative Commons Attribution License.

articles relating to correctness analysis of PLC programs [3], [6], [11], [12] is mainly devoted to construction of translators from IEC 61131 standard languages to interface languages of verification software. Demonstration of results is carried out on trivial examples. However, our experience of working with the practical logic control problems showed that the direct translation does nothing for analysis of program properties, since it is often not possible to express desired properties in temporal logic languages.

In this article, an approach to construction and verification of PLC-programs for discrete problems is proposed. For the specification of the program behavior, we use the linear-time temporal logic LTL. Programming is carried out in ST, IL and LD languages according to an LTL-specification. The correctness analysis of an LTL-specification is carried out by the symbolic model checking tool Cadence SMV. A new approach to programming and verification of PLC-programs is shown by an example. For a discrete problem we give an ST-program, its LTL-specification and an SMV-model. The purpose of the article is to describe an approach to programming PLC, which would provide a possibility of PLC-program correctness analysis by the model checking method.

The further work is to build software tools for modeling, specification, construction and verification of PLC-programs.

II. Model checking. A PLC program model

Model checking is the process of checking whether a given model (a Kripke structure) satisfies a given logical formula. A Kripke structure represents the behaviour of a program. A temporal logic formula encodes the property of the program. We use the linear-time temporal logic (LTL).

A Kripke Structure on a set of atomic propositions P is a state transition system $\mathcal{S} = (S, s_0, \to, L)$, with a non-empty set of states S, an initial state $s_0 \in S$, a transition relation $\to \subseteq S \times S$ which is defined for all $s \in S$, and a function $L : S \to 2^P$, labeling every state by a subset of atomic propositions.

A Path of the Kripke structure from the state s_0 — is an infinite consequence of states $\pi = s_0 s_1 s_2 \dots$ where $\forall\ i \geq 0$ $s_i \to s_{i+1}$.

The linear-time temporal logic language is considered as a specification language for behavioural properties of a programming model. PLC is a classic reactive control system, which once running must always have a correct infinite behavior. LTL formulas allow to represent this behavior.

The syntax of the LTL formula is given by the following grammar, $p_i \in P$:

$$\varphi, \psi ::= \mathit{true} \mid p_0 \mid p_1 \mid \dots \mid p_n \mid \neg \varphi \mid \psi \wedge \varphi \mid$$
$$X \varphi \mid \psi\, U\, \varphi \mid F\, \varphi \mid G\, \varphi.$$

LTL formula describes a property of one path of the Kripke structure, descendant from an emphasized current state. The temporal operators X, F, G and U are interpreted as follows: $X\varphi$ — φ must hold at the next state, $F\varphi$ — φ must hold at some future state, $G\varphi$ — φ must hold at the current state and all future state, $\psi\, U\, \varphi$ — φ holds at the current or a future state, and ψ must hold up until this point. In addition, classical logical operators $\vee$ and $\Rightarrow$ will be used further.

The Kripke structure satisfies an LTL formula (property) φ, if φ holds true for all paths, starting from the initial state s_0.

The Kripke model for a PLC program can be built quite naturally. For a state of the model we take a vector of values of all program variables, which can be divided into two parts. The first part is a value vector of inputs at the moment of the beginning of a new PLC working cycle. The second part is a value vector of outputs and internal variables after passing a complete working cycle (on the inputs from the first part). In other words, the state of the model is a state of a PLC-program after the complete passing of a working cycle. Thus, a transition from one state to another depends on the (previous) values of the outputs and internal variables of the first state and the (new) values of the inputs of the second state. For each state, the degree of the transition relation branching is determined by the number of all possible combinations of PLC input signals. Atomic propositions of the model are logical expressions on PLC program variables with using arithmetic and relational operators.

III. Programming concept

A purpose of the article is to describe an approach to programming PLC, which would provide a possibility of PLC-program correctness analysis by the model checking method. We will proceed from convenience and simplicity of using the model checking method. We require holding two following conditions.

Condition 1. The value of each variable must not change more than once per one full execution of the program while passing the PLC working cycle.

Condition 2. The value of each variable must change at only one place in the program in some operation block without nestings.

This conditions are reasonable assumption because inputs are always latched while operating the cycle. We will change the variable value only when it is really necessary, i. e. we will forbid an access to the variable by assigning if conditions of mandatory changing of its value do not hold. In this approach, the requirements of changing the value of a certain variable V after one pass of the PLC working cycle are represented by the following LTL formulas.

The next LTL formula is used for describing situations, leading to an increase of the variable value V

$$\mathbf{GX}(V > _V \Rightarrow \mathit{OldValCond} \wedge \mathit{FiringCond} \wedge V = \mathit{NewValExpr}) \quad (1)$$

This formula means that whenever a new value of variable V is larger than its previous value, recorded in the variable $_V$, it follows that the old value of variable V satisfies the condition $\mathit{OldValCond}$, a condition of the external action $\mathit{FiringCond}$ is accomplished, and the new value of variable V is the value of the expression $\mathit{NewValExpr}$.

The leading underscore symbol "_" in the denotation of the variable $_V$ is taken as a pseudo-operator, allowing to refer to

the previous state value of a variable V. This pseudo-operator can be used only under the scope of the temporal operator $\mathbf{X}$.

Conditions $FiringCond$ and $OldValCond$ are logical expressions on program variables and constants, which are constructed using comparison operators, logical and arithmetic operators and the pseudo-operator "_". By definition, the pseudo-operator can be applied only to variables. The expression $FiringCond$ describes situations, when changing the value of the variable V is needed (if it is allowed by the condition $OldValCond$). The expression $NewValExpr$ is built using variables and constants, comparison, logical and arithmetic operators and the pseudo-operator "_".

For descriptions of all possible increasing value situations the formula (1) may have several sets of considered conjunctive parts $OldValCond_i \wedge FiringCond_i \wedge V = NewValExpr_i$, combined in a disjunction, after the operator $\Rightarrow$.

Situations that lead to a decrease of V value are described similarly:

$$\mathbf{GX}(V < _V \Rightarrow OldValCond' \wedge FiringCond' \wedge V = NewValExpr') \quad (1')$$

Temporal formulas of the form (1) and $(1')$ describe a desired behavior of some integer variable. A more simple LTL formula is proposed to use in case of a logical (binary) data type variable. The following formula describes situations which increase the value of a binary variable V:

$$\mathbf{G}\,\mathbf{X}(\,\neg_V \wedge V \Rightarrow FiringCond\,). \quad (2)$$

Situations that lead to a decrease of the variable V value are described similarly:

$$\mathbf{G}\,\mathbf{X}(\,_V \wedge \neg V \Rightarrow FiringCond'\,). \quad (2')$$

Let's consider a special case of the specification form (1) and $(1')$, where for V we have
$FiringCond \ = FiringCond' = 1$,
$NewValExpr \ = NewValExpr'$,
$OldValCond \ = (_V < NewValExpr)$ and
$OldValCond' = (_V > NewValExpr)$:

$$\mathbf{G}\,\mathbf{X}(\,V > _V \Rightarrow _V < NewValExpr \wedge V = NewValExpr\,);$$

$$\mathbf{G}\,\mathbf{X}(\,V < _V \Rightarrow _V > NewValExpr \wedge V = NewValExpr\,).$$

This specification can be replaced by the following LTL formula:

$$\mathbf{G}\,\mathbf{X}(\,V = NewValExpr\,). \quad (3)$$

The variable V we will call a *register-variable*, if it has specification of forms (1), $(1')$, (2) and $(2')$. If V is constructed by specification of the form (3), it is called a *function-variable*. In the special case of specification (3), where the expression $NewValExpr$ does not contain leading underscore pseudo-operator "_", variable V is called a *substitution-variable*.

It is important to note that each LTL formula template is constructive, i. e. the program can be easily build from specification that would correspond to temporal properties expressed by these formulas. Thus, we can say that PLC programming is reduced to building a behavior specification of each program variable, which is output or auxiliary internal variable. The process of writing a program code is completed, when specification for each such variable is created. Note, that quantity and meaning of output variables are defined by a PLC and a problem formulation.

The program specification is divided into two parts: 1) specification of a behaviour of all program variables (except inputs), 2) specification of common program properties. The second part of specification affects quantity and meaning of internal auxiliary PLC program variables.

In specification it is important to consider the order of temporal formulas describing the behavior of the variables. A variable without the pseudo-operator "_" may be involved in the specification of another variable behavior only if the specification of its behavior is completed and is in the text above.

If necessary, we will use the keyword "Init" for indication of a variable initial value. For example, Init$(V) = 1$ means that the variable V initially is set to 1. If the initial value of some variable is not explicitly defined, it is assumed that this value is zero.

IV. Programming by specification

In this section, we consider a way of constructing a program code by constructive LTL-specification of the program variable behavior. In general, a translation process from LTL-formulas to program code is the following. Two temporal formulas of variable V, marked $V+$ (value increase, (1)) and $V-$ (value decrease, $(1')$), are set in conformity to the text block in the ST language:

```
IF      OldValCond AND FiringCond THEN
    V := NewValExpr;                              (* V+ *)
ELSIF OldValCond' AND FiringCond' THEN
    V := NewValExpr';                             (* V- *)
END_IF;
```

in the IL language:

```
        calculation of OldValCond AND FiringCond
        JMPCN   VL1
        calculation of NewValExpr
        ST      V
        JMP     VLEND
VL1:
        calculation of OldValCond' AND FiringCond'
        JMPCN   VL2
        calculation of NewValExpr'
        ST      V
        JMP     VLEND
VL2:
VLEND:
```

If the number of conjunctive blocks

$$OldValCond_i \wedge FiringCond_i \wedge V = NewValExpr_i$$

in LTL formulas will be more than two, the number of alternative branches "ELSIF" or labels will grow (by one branch or label for each new block).

Note, that the behavior of the obtained program will completely satisfy LTL formulas of specifications.

For LTL formula of a boolean variable V behavior in the forms $V+$ (2) and $V-$ (2′), we have the following ST-block:

```
IF NOT _V AND FiringCond THEN V := 1; (* V+ *)
ELSIF _V AND FiringCond′ THEN V := 0; (* V− *)
END_IF;
```

IL-block:

```
        calculation of NOT _V AND FiringCond
        JMPCN   VL1
        S       V
        JMP     VLEND
VL1:
        calculation of _V AND FiringCond′
        JMPCN   VL2
        R       V
        JMP     VLEND
VL2:
VLEND:
```

LD-block (in LD only boolean variable are used):

```
 _V                                     V
–|/|————————FiringCond ——————(S)—————-
 _V                                     V
–| |————————FiringCond′——————(R)—————-
```

Each program variable must be defined in the description section (local or global) and initialized in conformity with the specification. Note that, for example, in CoDeSys [8] all variables are initialized to zero by default.

In addition, we must implement the idea of the pseudo-operator "_". To do this, in the end of the program an area for a pseudo-operator section is allocated. In this area an assignment $_V := V$ is added after description of the behavior of all specification variables. In IL this assignment is:

```
LD      V
ST      _V
```

An assignment in LD:

```
      V                                   _V
———-| |———————————————————————( )——————————
```

The assignment is added for each variable V, to the last value of which is addressed as $_V$. The variable $_V$ is also necessary to define in the description section with the same initialization as for the variable V.

Note, that the approach to programming by specification, which describes the reason of changing each program variable value, looks very natural and reasonable, because a PLC output signal is the control signal, and changing the value of this control signal usually carries an additional meaning. For example, it is important to understand why an engine or some lamp must be turned on/off. Therefore, it seems quite obvious that every variable must be accompanied by two properties, one for each direction changing. It is assumed that if change conditions are not made, the variable remains at its previous state.

V. Building SMV-model by specification

We consider the verifier Cadence SMV [13] as a software tool of correctness analysis by model checking method. It is proposed to build a Kripke structure model in the SMV language with further verification of common program properties satisfiability for this model after creating the specification. If some common program property is not hold for the model, the verifier builds an example of incorrect path in a Kripke structure model, by which corrections in the specification are produced. And only after all the program properties have been verified with positive results, ST-progam of PLC is built by specification.

The SMV language allows to define a variable value in the next state of a model by using the "next" operator. Branching of the transition relation is provided by the "nondeterministic" assignment. For example, assignment next(V) := $\{0, 1\}$ means that states and transitions to them will be generated both with a value of $V = 0$, and with a value of $V = 1$. In the SMV language the symbols "&", "|", "∼" and "–>" denote logical "and", "or", "not" and implication, respectively.

The SMV language is oriented on creating the next states of Kripke models from the current state. The initial current state of the model is the state of program after initialization. Therefore, specification of the behavior of a variable V (1) and (1′) will be easier (clearer) to rewrite in the following equivalent form

$V+$: $\mathbf{G}(\,\mathbf{X}(V > _V) \Rightarrow$
$\mathbf{X}(\,\mathit{OldValCond}\,) \wedge \mathbf{X}(\mathit{FiringCond}\,) \wedge \mathbf{X}(V = \mathit{NewValExpr}\,)),$
$V-$: $\mathbf{G}(\,\mathbf{X}(V < _V) \Rightarrow$
$\mathbf{X}(\,\mathit{OldValCond}') \wedge \mathbf{X}(\mathit{FiringCond}') \wedge \mathbf{X}(V = \mathit{NewValExpr}')).$

And then we get an SMV-model of a variable V behavior quite naturally, putting the "next" operator in conformity to the temporal operator $\mathbf{X}$:

```
case{ next(OldValCond ) & next(FiringCond ) :
                                next(V) := next(NewValExpr );
      next(OldValCond′) & next(FiringCond′) :
                                next(V) := next(NewValExpr′);
      default :                 next(V) := V; }.
```

Keyword "default" means what must happen by default, i. e. if conditions of first two branches in the "case" block don't hold.

In the case of a boolean variable V specification (2) and (2′) is converted to the following SMV-model

```
case{ ∼V & next(FiringCond ) : next(V) := 1;
       V & next(FiringCond′) : next(V) := 0;
      default                : next(V) := V;}.
```

A model of a function-variable behavior is defined as next(V) := next($\mathit{NewValExpr}$).

Let's now consider a specification of the behavior of a substitution-variable V. In this case $\mathit{NewValExpr}$ does not contain pseudo-operator "_". This allows to rewrite the specification in the following equivalent form:

V: $\mathbf{X}\,\mathbf{G}(\,V = \mathit{NewValExpr}\,).$

In fact, this formula means that if the initial state of the model does not considered, then an equality $V = \mathit{NewValExpr}$ must hold in all other states of the model. Fairness of formula $\mathbf{X}\,\mathbf{G}(\,V = \mathit{NewValExpr}\,)$ follows from the fairness of more

general formula $\mathbf{G}(\,V = NewValExpr\,)$. Therefore, more general formula can be used as the constructive specification for building SMV-model of a substitution-variable V. SMV-model is built by this specification in the form of assignment

$V := NewValExpr$.

The Cadence SMV verifier allows to check program models, containing up to 59 binary variables (all variables in SMV are represented by sets of binary variables). The substitution-variables are not included in this number, i. e. only register-variables and function-variables are considered.

VI. Conclusion

The approach has been successfully approved on some (about a dozen) "discrete" logical control problems of different types with the average number of binary PLC inputs and outputs about 30 and the total number of binary program variables up to 59. For example, properties, relating to maintenance of the technological process (to exclude the possibility of nonconforming product outflow), were verified for a PLC program, controlling a mixture preparation device. Properties relating to connection standby pumps in time were tested for problem of hydraulic pump system control. And properties of mandatory execution of received commands of cabin lift calling were tested for library lift control problem. The verification work was carried out on PC with processor Intel Core i7 2600K 3.40 GHz. Time spent by verifier Cadence SMV to check specified properties is limited to a few seconds.

The further work is to build software tools for modeling, specification, construction and verification of PLC-programs, according to the results of the work on this topic.

Acknowledgements

This work was financially supported by the Russian Foundation for Basic Research (project no. 12-01-00281-a).

References

[1] *Kuzmin E. V.*, *Sokolov V. A.* Modeling, Specification and Construction of PLC-programs // Modeling and analysis of information systems. 2013. V. 20, No. 2. P. 104–120 [in Russian].

[2] *Kuzmin E. V.*, *Sokolov V. A.* On Construction and Verification of PLC-programs // Modeling and analysis of information systems. 2012. V. 19, No. 4. P. 25–36. [In Russian].

[3] *Kuzmin E. V.*, *Sokolov V. A.* On Verification of PLC-programs Written in the LD-Language // Modeling and analysis of information systems. 2012. V. 19, No. 2. P. 138–144. [In Russian]

[4] *Kuzmin E. V.*, *Sokolov V. A.*, *Ryabukhin D. A.* Construction and Verification of PLC LD-programs by LTL-specification // Modeling and analysis of information systems. 2013. V. 20, No. 6. P. 78–94 [in Russian].

[5] *Petrov I. V.* Programmiruemye kontrollery. Standartnye jazyki i priemy prikladnogo proektirovanija. M.: SOLON-Press, 2004. 256 p. [In Russian].

[6] *Canet G.*, *Couffin S.*, *Lesage J.-J.*, *Petit A.*, *Schnoebelen Ph.* Towards the Automatic Verification of PLC Programs Written in Instruction List // Proc. of the IEEE International Conference on Systems, Man and Cybernetics. Argos Press, 2000. P. 2449–2454.

[7] *Clark E. M.*, *Grumberg O.*, *Peled D. A.* Model Checking. The MIT Press, 2001.

[8] CoDeSys. Controller Development System. http://www.3s-software.com/

[9] *Gries D.* The Science of Programming. Springer-Verlag, 1981.

[10] *Parr E. A.* Programmable Controllers. An engineer's guide. Newnes, 2003. 442 p.

[11] *Pavlovic O.*, *Pinger R.*, *Kollman M.* Automation of Formal Verification of PLC Programs Written in IL // Proc. of 4th International Verification Workshop (VERIFY'07). Bremen, Germany, 2007. P. 152–163.

[12] *Rossi O.*, *Schnoebelen Ph.* Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs // Proc. of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, Shaker Verlag, 2000. P. 177–182.

[13] SMV (Symbolic Model Verifier). The Cadence SMV Model Checker. http://www.kenmcmil.com/smv.html

Appendix A

Library lift

A library lift scheme is represented in Fig. 1. The purpose of the lift is to lift up books on request from the basement to the first and second floors of the library and to return them to the basement.

The elevator cabin is called from the base floor by pressing buttons "Up 2" and "Up 1". If the corresponding command was accepted, this button lamp turns on. It turns off when the command is done. When the cabin is on some floor, the lamps "Floor 2", "Floor 1" or "Floor 0" are on. There are shaft doors, which are opened and closed manually. The sensors "DS2", "DS1" and "DS0" are needed to determine the position of doors. The door sensor is on if door is closed. The signal from the door sensor is determined using a lamp of the sensor.

The floor sensor "FS" in the elevator cabin is used for finding the position of the cabin in the shaft. The floor sensor is on, if the cabin is entirely on a particular floor. Otherwise, the signal is removed.

The library lift control is carried out using a PLC receiving input signals from sensors and buttons and sending output signals to the lift motor and lamps. The task is to construct a PLC program with 10 binary inputs and 14 binary outputs for controling the lift. PLC interface is shown in Fig. 2. More detailed requirements for the library lift program are given in the article [4]. The constructive specification of the library lift control program was built according to these requirements and the programming concept by LTL-specification.

```
Ctr0+:  GX(~_Ctr0 &  Ctr0 -> FS & ~_FS &
                    _Ctr1 & ~_Dir);
Ctr0-:  GX( _Ctr0 & ~Ctr0 -> FS & ~_FS );
Ctr1+:  GX(~_Ctr1 &  Ctr1 -> FS & ~_FS &
                    (_Ctr2 & ~_Dir | _Ctr0 & _Dir));
Ctr1-:  GX( _Ctr1 & ~Ctr1 -> FS & ~_FS );
init(Ctr1)=1;
Ctr2+:  GX(~_Ctr2 &  Ctr2 -> FS & ~_FS &
                    _Ctr1 & _Dir);
Ctr2-:  GX( _Ctr2 & ~Ctr2 -> FS & ~_FS );
Up01+:  GX(~_Up01 &  Up01 -> FS & Ctr0 & PBUp01);
Up01-:  GX( _Up01 & ~Up01 -> FS & Ctr1 & ~_Mtr );
Up02+:  GX(~_Up02 &  Up02 -> FS & Ctr0 & PBUp02);
Up02-:  GX( _Up02 & ~Up02 -> FS & Ctr2 & ~_Mtr );
Dwn2+:  GX(~_Dwn2 &  Dwn2 -> FS & Ctr2 & ~_Mtr &
                    (PBDwn2 | _Tmr.Q));
Dwn2-:  GX( _Dwn2 & ~Dwn2 -> FS & Ctr0 & ~_Mtr);
Dwn1+:  GX(~_Dwn1 &  Dwn1 -> FS & Ctr1 & ~_Mtr &
                    (PBDwn1 | _Tmr.Q));
Dwn1-:  GX( _Dwn1 & ~Dwn1 -> FS & Ctr0 & ~_Mtr );
Init(Dwn1)=1;
Flr2+:  GX(~_Flr2 &  Flr2 -> PBFlr2);
Flr1+:  GX(~_Flr1 &  Flr1 -> PBFlr1);
Flr2-:  GX( _Flr2 & ~Flr2 -> FS & Ctr2 & ~_Mtr);
```

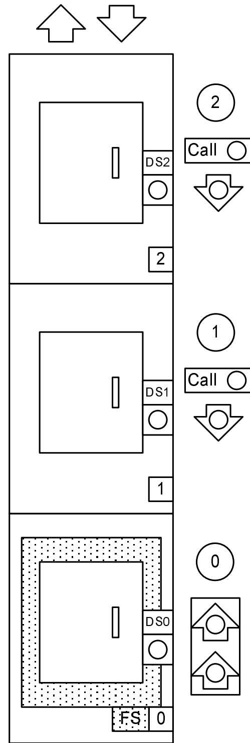


Fig. 1. A scheme of a library lift

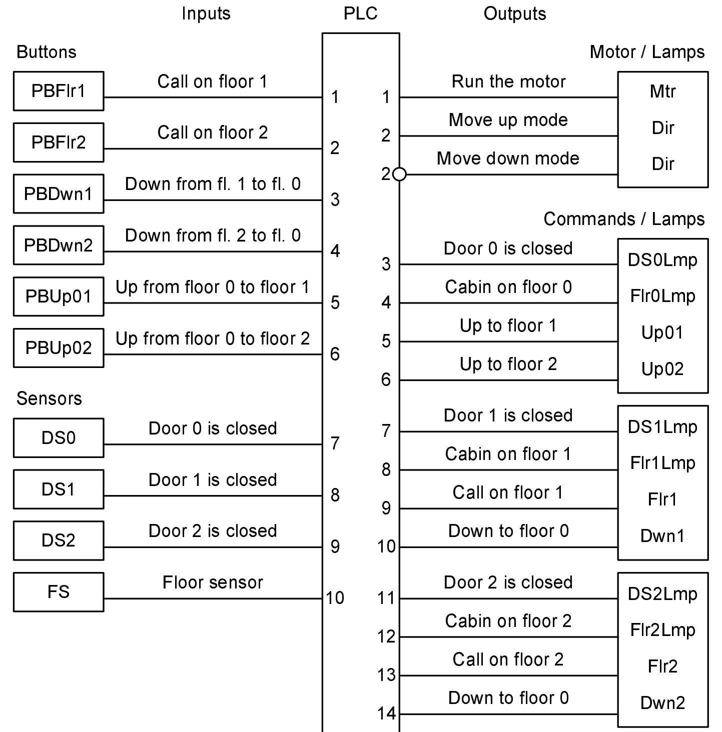


Fig. 2. PLC control interface of a library lift

```
Flr1 -: GX( _Flr1 & ~Flr1 -> FS & Ctr1 & ~_Mtr);
Dir+: GX(~_Dir &  Dir -> FS & Ctr0 & ~_Mtr);
Dir-: GX( _Dir & ~Dir -> (Ctr2 | Ctr1 & Dwn1 &
        ~Up02 & (~Flr2 | Dwn2)) & FS & ~_Mtr);
DS: GX(DS = DS0 & DS1 & DS2);
Mtr+: GX(~_Mtr &  Mtr -> DS &
  ~FS & (Dwn1 | Dwn2 | Up01 | Up02 | Flr1 | Flr2) |
   FS & Ctr0 & (Up01 | Up02 | _Tmr.Q & (Flr1|Flr2))|
   FS & Ctr1 & Dwn1 | FS & Ctr2 & Dwn2));
Mtr-: GX( _Mtr & ~Mtr -> (~DS |
                 FS & Ctr0 & ~_Dir |
                 FS & Ctr2 &  _Dir |
                 FS & Ctr1 & (Flr1 & ~Dwn1 | Up01)));
Tmr.In: GX(Tmr.In  = ~Mtr & FS &
        (Ctr0 & DS0 | Ctr1 & DS1 | Ctr2 & DS2));
Flr0Lmp: GX(Flr0Lmp = ~Mtr & FS & Ctr0);
DS0Lmp: GX(DS0Lmp = DS0);
Flr1Lmp: GX(Flr1Lmp = ~Mtr & FS & Ctr1);
DS1Lmp: GX(DS1Lmp = DS1);
Flr2Lmp: GX(Flr2Lmp = ~Mtr & FS & Ctr2);
DS2Lmp: GX(DS2Lmp = DS2).
```

This specification allows to check the following common program properties of the library lift control program.

1. $\mathbf{G}(\mathit{Ctr0} + \mathit{Ctr1} + \mathit{Ctr2} = 1)$ means that at any moment the elevator cabin is only on one floor.

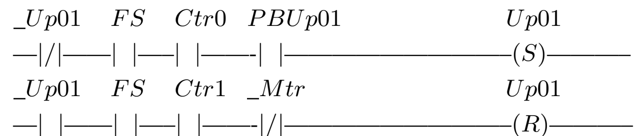
2. There are no situations, when the elevator cabin is on the basement and the motor is in descent mode: $\mathbf{G}\neg(\mathit{FS} \wedge \mathit{Ctr0} \wedge \mathit{Dir}=0 \wedge \mathit{Mtr})$. There are no situations, when the elevator cabin is on the second floor and the motor is in ascent mode: $\mathbf{G}\neg(\mathit{FS} \wedge \mathit{Ctr2} \wedge \mathit{Dir}=1 \wedge \mathit{Mtr})$.

3. Always, when the motor is on, the shaft doors are closed: $\mathbf{G}(\mathit{Mtr} \Rightarrow \mathit{DS})$. And if the shaft doors are closed and the cabin is not entirely on a particular floor, the motor is on: $\mathbf{G}(\mathit{DS} \wedge \neg \mathit{FS} \Rightarrow \mathit{Mtr})$. There are no situations when the motor is off, the doors are closed and the cabin is not entirely on a particular floor: $\mathbf{G}\neg(\neg \mathit{Mtr} \wedge \mathit{DS} \wedge \neg \mathit{FS})$.

4. Every moment when motor turns on, it shall be turned off in future: $\mathbf{G}(\mathit{Mtr} \Rightarrow \mathbf{F}(\neg \mathit{Mtr}))$.

5. If these program executions are considered, when after receipt of a call/send to a floor command Cmd and before its execution, the shaft door will be opened or closed only a finite number of times. Always in this case Cmd sooner or later will be executed, where Cmd — is $\mathit{Flr1}$, $\mathit{Flr2}$, $\mathit{Up01}$, $\mathit{Up02}$, $\mathit{Dwn1}$ or $\mathit{Dwn2}$: $\mathbf{G}(\mathit{Cmd} \Rightarrow \neg\mathbf{G}(\mathit{Cmd}\ \mathbf{U}\ \neg \mathit{DS})) \Rightarrow \mathbf{G}(\mathit{Cmd} \Rightarrow \mathbf{F}(\neg \mathit{Cmd}))$.

For the variable $\mathit{Up01}$ we give an example of its IL and LD code blocks built by the LTL specification. LD-block:



IL-block:

```
          LDN     _Up01     (* Up01+ *)
          AND     FS
          AND     Ctr0
          AND     PBUp01
          JMPCN   Up01L1
          S       Up01
          JMP Up01LEND
Up01L1:   LD      _Up01     (* Up01- *)
          AND     FS
          AND     Ctr1
          ANDN    _Mtr
```

```
            JMPCN   Up01LEND
            R       Up01
            JMP     Up01LEND
Up01L2:
Up01LEND:
```

ST-program is built by constructive LTL-specification after checking common program properties.

SMV-model of the "library lift" program

Modeling of a timer is described in [1]. A model of behaviour of floor sensor FS is following. When the motor is on, a value of the sensor FS can be changed. When the motor is off, a value of the sensor FS remains unchanged. Fairness conditions for FS mean that when the motor is on, floor sensor can not remain unchanged indefinitely.

```
module timer(){
  I : 0..1;    /* Input  */
  Q : 0..1;    /* Output */
  init(I):=0; init(Q):=0;
  next(Q):= next(I) & (Q | {0, 1});
  FAIRNESS I -> Q;
}
module main(){    /* Inputs */
/* Buttons  */
  PBFlr2,PBDwn2,PBFlr1,PBDwn1,PBUp02,PBUp01: 0..1;
/* Sensors */
  DS2, DS1, DS0, FS: 0..1;
/* Outputs */
  Mtr, Dir: 0..1; /* Motor  */
  Flr2Lmp, Flr1Lmp, Flr0Lmp, DS2Lmp, DS1Lmp,
  DS0Lmp: 0..1;    /* Lamps   */
/* Commands */
  Flr2, Flr1, Dwn2, Dwn1, Up02, Up01: 0..1;
/* Auxiliary */
  Ctr0, Ctr1, Ctr2: 0..1;
  DS: 0..1;
  Tmr : timer;
/*Initialization section */
/* Inputs */
  init(PBFlr2):=0; init(PBFlr1):=0;
  init(PBUp02):=0; init(PBUp01):=0;
  init(PBDwn2):=0; init(PBDwn1):=0;
  init(DS1):=0; init(DS0):=0;
  init(DS2):=0; init(FS):=0;
/* Outputs */
  init(Mtr):=0; init(Flr1):=0; init(Dwn1):=1;
  init(Up01):=0; init(Dir):=0; init(Flr2):=0;
  init(Dwn2):=0; init(Up02):=0;
/* Auxiliary */
  init(Ctr0):=0; init(Ctr1):=1; init(Ctr2):=0;
/* Transition system */
/* Inputs */
  next(PBFlr2):={0, 1}; next(PBDwn2):={0, 1};
  next(PBFlr1):={0, 1}; next(PBDwn1):={0, 1};
  next(PBUp02):={0, 1}; next(PBUp01):={0, 1};
  next(DS2):={0, 1}; next(DS1):={0, 1};
  next(DS0):={0, 1};
  case{ Mtr     : next(FS):={0,1};
        default : next(FS):=FS;};
  FAIRNESS Mtr -> FS;
  FAIRNESS Mtr -> ~FS;
/* Outputs and auxiliary */
case{~Ctr0 & next(FS) & ~FS & Ctr1 & ~Dir :
        next(Ctr0):=1;                     /* Ctr0+ */
  Ctr0 & next(FS) & ~FS :
        next(Ctr0):=0;                     /* Ctr0- */
  default : next(Ctr0):=Ctr0;   };
case{~Ctr1 & next(FS) & ~FS &
  (Ctr2 & ~Dir | Ctr0 & Dir) :
        next(Ctr1):=1;                     /* Ctr1+ */
  Ctr1 & next(FS) & ~FS :
        next(Ctr1):=0;                     /* Ctr1- */
  default : next(Ctr1):=Ctr1;};
case{~Ctr2 & next(FS) & ~FS & Ctr1 &  Dir:
        next(Ctr2):=1;                     /* Ctr2+ */
  Ctr2 & next(FS) & ~FS :
        next(Ctr2):=0;                     /* Ctr2- */
  default : next(Ctr2):=Ctr2;};

case{~Up01 & next(FS) & next(Ctr0) & next(PBUp01):
        next(Up01):=1;                     /* Up01+ */
  Up01 & next(FS) & next(Ctr1) & ~Mtr:
        next(Up01):=0;                     /* Up01- */
  default : next(Up01):=Up01;};
case{~Up02 & next(FS) & next(Ctr0) & next(PBUp02):
        next(Up02):=1;                     /* Up02+ */
  Up02 & next(FS) & next(Ctr2) & ~Mtr:
        next(Up02):=0;                     /* Up02- */
  default : next(Up02):=Up02;};
case{~Dwn2 & next(FS) & next(Ctr2) &
  ~Mtr & (next(PBDwn2) | Tmr.Q):
        next(Dwn2):=1;                     /* Dwn2+ */
  Dwn2 & next(FS) & next(Ctr0) & ~Mtr:
        next(Dwn2):=0;                     /* Dwn2- */
  default : next(Dwn2):=Dwn2;};
case{~Dwn1 & next(FS) & next(Ctr1) &
  ~Mtr & (next(PBDwn1) | Tmr.Q):
        next(Dwn1):=1;                     /* Dwn1+ */
  Dwn1 & next(FS) & next(Ctr0) & ~Mtr:
        next(Dwn1):=0;                     /* Dwn1- */
  default : next(Dwn1):=Dwn1;};
case{~Flr2 & next(PBFlr2):
        next(Flr2):=1;                     /* Flr2+ */
  Flr2 & next(FS) & next(Ctr2) & ~Mtr:
        next(Flr2):=0;                     /* Flr2- */
  default  : next(Flr2):=Flr2;};
case{~Flr1 & next(PBFlr1):
        next(Flr1):=1;                     /* Flr1+ */
  Flr1 & next(FS) & next(Ctr1) & ~Mtr:
        next(Flr1):=0;                     /* Flr1- */
  default : next(Flr1):=Flr1;};
case{~Dir & next(FS) & next(Ctr0) & ~Mtr:
        next(Dir):=1;                      /* Dir+ */
  Dir & next(FS) & (next(Ctr2) | next(Ctr1) &
  next(Dwn1) & ~next(Up02) & (~next(Flr2) |
  next(Dwn2))) & ~Mtr:
        next(Dir):=0;                      /* Dir- */
  default  : next(Dir):=Dir;};
  DS:= DS0 & DS1 & DS2;                    /*  DS  */
case{~Mtr & next(DS) &
  (~next(FS) & (next(Dwn1)|next(Dwn2)|next(Up01)|
  next(Up02)|next(Flr1)|next(Flr2))|
  next(FS) & next(Ctr0) & (next(Up01)|next(Up02)|
  Tmr.Q & (next(Flr1) | next(Flr2))) |
  next(FS) & next(Ctr1) & next(Dwn1) |
  next(FS) & next(Ctr2) & next(Dwn2)):
        next(Mtr):=1;                      /* Mtr+ */
  Mtr &(~next(DS) | next(FS) & next(Ctr0) & ~Dir |
  next(FS) & next(Ctr2) &  Dir | next(FS) &
  next(Ctr1) & (next(Flr1) &~next(Dwn1)|next(Up01))):
        next(Mtr):=0;                      /* Mtr- */
  default : next(Mtr):=Mtr;};
  next(Tmr.I):= next(~Mtr & FS &
    (Ctr0 & DS0 | Ctr1 & DS1 | Ctr2 & DS2));
  Flr0Lmp:= ~Mtr & FS & Ctr0;          /* Flr0Lmp */
  Flr1Lmp:= ~Mtr & FS & Ctr1;          /* Flr1Lmp */
  Flr2Lmp:= ~Mtr & FS & Ctr2;          /* Flr2Lmp */
  /* DS0Lmp, DS1Lmp, DS2Lmp   */
  DS0Lmp:= DS0; DS1Lmp:= DS1; DS2Lmp:= DS2;
/* Properties section */
  P_Ctr: assert G(Ctr0+Ctr1+Ctr2 = 1);
  P_Limit0: assert G~(FS & Ctr0 & Dir=0 & Mtr);
  P_Limit2: assert G~(FS & Ctr2 & Dir=1 & Mtr);
  P_Doors: assert G(Mtr -> DS);
```

```
  P_Stop: assert G~(~Mtr & DS & ~FS);
  P_Mtr: assert G( Mtr -> F(~Mtr));
  P_Flr2: assert G(Flr2 -> ~G(Flr2 U ~DS)) ->
    G(Flr2 -> F(~Flr2));
  P_Flr1: assert G(Flr1 -> ~G(Flr1 U ~DS)) ->
    G(Flr1 -> F(~Flr1));
  P_Up01: assert G(Up01 -> ~G(Up01 U ~DS)) ->
    G(Up01 -> F(~Up01));

  P_Up02: assert G(Up02 -> ~G(Up02 U ~DS)) ->
    G(Up02 -> F(~Up02));
  P_Dwn1: assert G(Dwn1 -> ~G(Dwn1 U ~DS)) ->
    G(Dwn1 -> F(~Dwn1));
  P_Dwn2: assert G(Dwn2 -> ~G(Dwn2 U ~DS)) ->
    G(Dwn2 -> F(~Dwn2));
  P_Move: assert G(DS & ~FS -> Mtr);
}
```

ST-program of “library lift”

```
VAR_GLOBAL
(* Inputs*)
  PBFlr2,PBDwn2,PBFlr1, PBDwn1, PBUp02,PBUp01: BOOL;
  DS2, DS1, DS0, FS: BOOL;
(* Outputs*)
  Mtr, Dir, Flr2Lmp, Flr1Lmp, Flr0Lmp, DS2Lmp: BOOL;
  DS1Lmp, DS0Lmp, Flr2,Flr1, Dwn2, Up02, Up01: BOOL;
  Dwn1: BOOL :=1;
END_VAR

PROGRAM PLC_PRG
VAR
  Tmr: TON := (PT := T#10s);
  Ctr0, Ctr2: BOOL;
  Ctr1: BOOL := TRUE;
  _Ctr, _FS, _Dir, _Dwn1, _Dwn2, _Flr1, _Flr2: BOOL;
  _Mtr, _TmrQ, _Up01, _Up02: BOOL;
END_VAR
IF NOT _Ctr0 AND FS AND NOT _FS AND
        _Ctr1 AND NOT _Dir THEN
        Ctr0:=1;                          (* Ctr0+ *)
ELSIF  _Ctr0 AND FS AND NOT _FS THEN
        Ctr0:=0;                          (* Ctr0- *)
END_IF;
IF NOT _Ctr1 AND FS AND NOT _FS AND
        (_Ctr2 AND NOT _Dir OR
        _Ctr0 AND _Dir) THEN
        Ctr1:=1;                          (* Ctr1+ *)
ELSIF  _Ctr1 AND FS AND NOT _FS THEN
        Ctr1:=0;                          (* Ctr1- *)
END_IF;
IF NOT _Ctr2 AND FS AND NOT _FS AND
        _Ctr1 AND _Dir THEN
        Ctr2:=1;                          (* Ctr2+ *)
ELSIF  _Ctr2 AND FS AND NOT _FS THEN
        Ctr2:=0;                          (* Ctr2- *)
END_IF;
IF NOT _Up01 AND FS AND Ctr0 AND PBUp01   THEN
        Up01:=1;                          (* Up01+ *)
ELSIF  _Up01 AND FS AND Ctr1 AND NOT _Mtr THEN
        Up01:=0;                          (* Up01- *)
END_IF;
IF NOT _Up02 AND FS AND Ctr0 AND PBUp02   THEN
        Up02:=1;                          (* Up02+ *)
ELSIF  _Up02 AND FS AND Ctr2 AND NOT _Mtr THEN
        Up02:=0;                          (* Up02- *)
END_IF;
IF NOT _Dwn2 AND NOT _Mtr AND FS AND Ctr2 AND
        (PBDwn2 OR _TmrQ) THEN Dwn2:=1;   (* Dwn2+ *)
ELSIF  _Dwn2 AND NOT _Mtr AND FS AND Ctr0 THEN
        Dwn2:=0;                          (* Dwn2- *)
END_IF;
IF NOT _Dwn1 AND NOT _Mtr AND FS AND Ctr1 AND
        (PBDwn1 OR _TmrQ) THEN Dwn1:=1;   (* Dwn1+ *)
ELSIF  _Dwn1 AND NOT _Mtr AND FS AND Ctr0 THEN
        Dwn1:=0;                          (* Dwn1- *)
END_IF;
IF NOT _Flr2 AND PBFlr2 THEN Flr2:=1;     (* Flr2+ *)
ELSIF  _Flr2 AND NOT _Mtr AND FS AND Ctr2 THEN
        Flr2:=0;                          (* Flr2- *)
END_IF;

IF NOT _Flr1 AND PBFlr1THEN Flr1:=1;      (* Flr1+ *)
ELSIF  _Flr1 AND NOT _Mtr AND FS AND Ctr1 THEN
        Flr1:=0;                          (* Flr1- *)
END_IF;
IF NOT _Dir AND NOT _Mtr AND FS AND Ctr0 THEN
        Dir:=1;                           (*  Dir+ *)
ELSIF  _Dir AND NOT _Mtr AND FS AND
        (Ctr2 OR Ctr1 AND Dwn1 AND NOT Up02 AND
        (NOT Flr2 OR Dwn2)) THEN Dir:=0;  (*  Dir- *)
END_IF;
DS:=DS0 AND DS1 AND DS2;

IF NOT _Mtr AND DS AND
        (NOT FS AND (Dwn1 OR Dwn2 OR
        Up01 OR Up02 OR Flr1 OR Flr2) OR
        FS AND Ctr0 AND
        (Up01 OR Up02 OR _TmrQ AND
        (Flr1 OR Flr2)) OR
        FS AND Ctr1 AND Dwn1 OR
        FS AND Ctr2 AND Dwn2) THEN Mtr:=1;(* Mtr+ *)
ELSIF _Mtr AND (NOT DS OR
        FS AND Ctr0 AND _Dir=0 OR
        FS AND Ctr2 AND _Dir=1 OR
        FS AND Ctr1 AND
        (Flr1 AND NOT Dwn1 OR Up01))
        THEN Mtr:=0;                      (* Mtr- *)
END_IF;
Tmr.In:= NOT Mtr AND FS AND
  (Ctr0 AND DS0 OR Ctr1 AND DS1 OR Ctr2 AND DS2);
Tmr();
Flr0Lmp:= NOT Mtr AND FS AND Ctr0;
Flr1Lmp:= NOT Mtr AND FS AND Ctr1;
Flr2Lmp:= NOT Mtr AND FS AND Ctr2;
DS0Lmp:=DS0;
DS1Lmp:=DS1;
DS2Lmp:=DS2;
(* -------- pseudo-operator section --------*)
_TmrQ:=Tmr.Q; _FS:=FS; _Mtr:=Mtr;
_Dir:=Dir; _Ctr:=Ctr;
_Dwn1:=Dwn1; _Dwn2:=Dwn2;
_Flr1:=Flr1; _Flr2:=Flr2;
_Up01:=Up01; _Up02:=Up02;
```