An Approach Using the B Method to Formal Verification of PLC Programs in an Industrial Setting*

Haniel Barbosa and David Déharbe

Departamento de Informática e Matemática Aplicada, UFRN, Brazil
hanielbbarbosa@gmail.com, deharbe@dimap.ufrn.br

Abstract. This paper presents an approach to verify PLCs, a common platform to control systems in the industry. We automatically translate PLC programs written in the languages of the IEC 61131-3 standard to B models, amenable to formal analysis of safety constraints and general structural properties of the application. This approach thus integrates formal methods into existing industrial processes, increasing the confidence in PLC applications, nowadays validated mostly through testing and simulation. The transformation from the PLC programs to the B models is described in detail in the paper. We also evaluate the approach's potential with a case study in a real railway application.

Keywords: B method, PLC, IEC 61131-3, safety critical systems, formal methods.

1 Introduction

Programmable Logic Controllers (from now on, PLCs) perform control operations in a system, running in *execution cycles*: they receive information from the environment as *inputs*, process them and affect this environment with the resulting *outputs*.

In many industries, such as mass transport and energy, it is very common to use PLCs in control applications. Those applications are mostly programmed according to IEC 61131-3 [1], an international standard that specifies the five standard PLC programming languages, namely: LD (Ladder Diagram) and FBD (Function Block Diagram), graphical languages; IL (Instruction List) and ST (Structured Text), textual languages; and SFC (Sequential Function Chart), that shows the structure and internal organization of a PLC. It is not rare that a variation of such languages is employed too.

As the complexity of the PLC applications increases, and as various are safety critical, it is important to ensure their reliability [2]. Since testing and simulation, the *de-facto* method in many branches to perform verification, can leave flaws

* Project supported by ANP. CNPq grants 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

undiscovered, something intolerable in safety-critical systems, another strategy is necessary. A mean to fulfill this requirement is with formal methods. However, they are difficult to integrate with the industrial process [3], since most control engineers are not familiarized with formal verification [4].

Some recent works have been trying to integrate formal methods and PLC programs verification, using different approaches. In [5], the authors created a new language combining ST and Linear Temporal Logic, ST-LTL, to ease the use of formal verification by control engineers. A method is presented in [6] to verify applications using Safety Function Blocks with timed-automata through model-checking and simulation. A model-driven engineering approach is used in [7] to generate models in a FIACRE language from LD programs. To this date, these approaches are concerned only with parts of the IEC 61131-3 standard.

Our approach consists of developing a tool that receives a PLC program based in the IEC 61131-3 standard and builds an intermediary model from it. This model is automatically translated to a formal model in the B notation [8]. Additional *safety constraints* requirements are then manually inserted and verified using theorem proving, thus avoiding state-explosion problems. We can also specify and verify dynamic properties, such as *deadlock freedom*, performing *model checking* in the model using the tool ProB[1], which also supports the definition and verification of new constraints in Linear Temporal Logic. Our approach is thus able to verify that the PLC is presenting the expected behavior in its *execution cycle*.

We chose the B Method because it is used successfully in safety-critical applications, e.g. in the railway industry [12]. Besides, it has a strong support of tools and the B language can handle decomposition, refinement and generation of verified code. It is better discussed in 2.2.

In order to include all the IEC 61131-1 languages, we based our intermediary model (from now on called "PLC model") in the PLCopen [9] standard, which provides an interface representing all such languages in an XML-based format, working also as *documentation*. This PLC model stands between the PLC programs and the formal models to be generated, then reducing the semantic gap between PLC and B and defining a unique semantics for different PLC languages [7]. The process also involves a customizable parser, so we can treat PLC programs that are not *strictly* following the IEC standard; as numerous legacy systems deviate from the standard, still our approach would thus be able to handle them.

Thus, as the generation of the formal model is automatic and as it makes correctness, according to the specification, a realistic and achievable goal, we facilitate the use of formal methods in industry and increase confidence in the PLC applications.

We also present a case study in a real safety-critical railway system: the Central Door Controller (from now on, CDC) of the doors subsystem of a train. We show the step by step automatic generation of the formal specification from its PLC program and, after defining the safety constraints, perform a full formal verification in the application, at the end exhibiting the results.

[1] `http://www.stups.uni-duesseldorf.de/ProB`

This paper presents the continuation of the work in [10] and [11]. A new definition of the formal model generated, as well as improvements on *how* it is generated and an evaluation of the whole method in a real case study are the main contributions of this new paper.

Structure of the paper. Section 2 presents in more detail PLCs and the B method. In section 3 we have the description of the different phases of our method, and in section 4 we present our case study. Section 5 concludes the paper and presents future work.

2 Context and Techniques

2.1 Programmable Logic Controllers

We base our work with PLCs on the PLCopen standard. This standard is an effort to gather all the information of the five different languages of the IEC standard and provide an interface with their supporting tools, as well as the ability to transfer information between different platforms. It is an XML- based standard able to store not just the textual, but also the graphical information of a project, allowing complete translation from a representation to another.

The PLCopen standard structures PLCs in three specific parts: the *Headers* structures, containing information such as the project name, the company associated, etc.; the *Instance* specific part, representing the configurations of the environment in which the PLC may operate; and the *Type* specific part, where we have the *Program Organization Units* (POUs) and the defined *Data Types.* In our approach we will consider only the elements of *Type.*

The Data Types are either *elementary* (Bool, Integer, etc.), *derived* (Enumeration, Array, Structure, etc.) or *extended* (Pointers); generic data types can also be defined. They are used to type the variables used in the POUs.

The POUs represent the PLC programs, being divided in three categories: *functions*, *function blocks* and *programs*:

- The POU *functions*, when executed, produce exactly *one* data statement – a variable typed according to one of the possible data types in the standard –, the function result, and arbitrarily many additional output variables. These POUs are stateless: they contain no state information, i.e., invocation of a function with the *same arguments* shall always produce the *same result.*
- The POU *function blocks* produce *one or more* data statements as result. The state of a function block persists from one execution to the next – they are stateful – , therefore invocation with the the *same arguments* may produce *different results.*
- The POU *programs* are defined as a “logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system”. Their declaration and usage is equivalent to the *function blocks.* It also may use the previous two POU types as auxiliary elements.

The three POU elements have an *interface* with its several kinds of variables: input, local, output, inout, etc. They also have a *body*, composed by IL (Instruction List), ST (Structured Text), LD (Ladder Diagram), FBD (Function Block Diagram) or SFC (Sequential Function Chart) elements, according to the language of the POU. In figure 1 we can see an example of a POU *program* in LD that makes use of an instantiation of a POU *function block* in FBD. In section 4 bits of PLC programs in SFC and ST are shown as part of our case study. For more details, see [1] and [9].

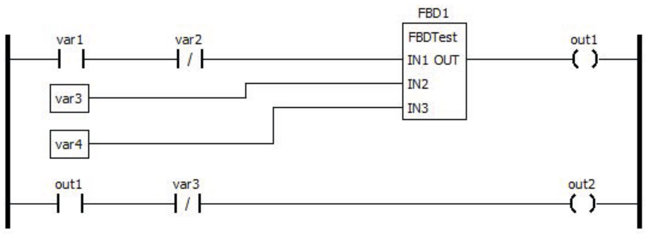


Fig. 1. LD program with rungs executed from left to right, sequentially. Boolean variables and a function block are evaluated in the execution.

2.2 B Method

The B Method [8] is a formal approach for the specification and development of software. It includes a first-order logic with integers and sets, substitutions and refinement rules. It is based on the Abstract Machine Notation (AMN), which provides a unique language for the construction of machines, refinements and implementations, thus representing the different levels of abstraction that a specification of a system may take. Besides, the language supports decomposition, since it is based around the concept of layered development and the construction of larger components from collections of smaller ones.

The B Method provides a unified pragmatic and usable development methodology based on the concept of refinement, requiring the consistency verification for each transformation of the specification from the abstract level towards the concrete one. This, along with the generation and verification of proof obligations to guarantee the consistency of the initial model, makes correctness according to the specification a realistic and achievable goal throughout system development.

3 The Method

The method we are proposing consists of three main phases:

1. translate the information in the PLC programs into an intermediary model, either from a standard or hybrid PLC program, or from an XML file in the PLCopen standard;

2. generate from it a B model that makes possible to check the structural and safety properties of the project;
3. and at last complete the formal model with such safety properties, derived from the project requirements (manually, for now).

Figure 2 illustrates the method. A case study covering all the phases of this method is shown in section 4.

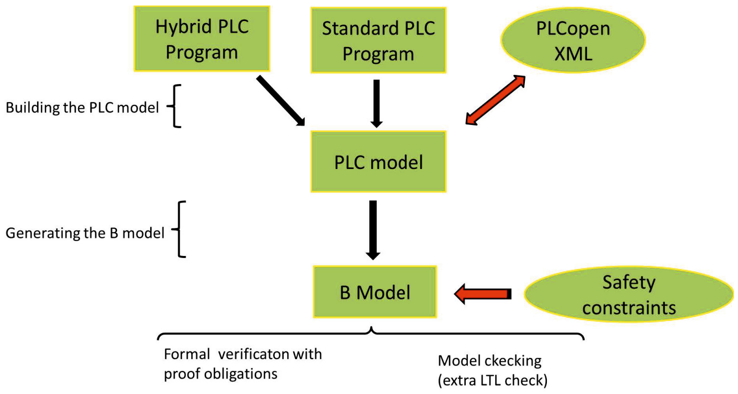


Fig. 2. Illustration of the complete method

3.1 Towards the PLC Model

The PLC model may be generated either directly from an PLCopen XML-based representation, from the programs in the standard languages or from programs in some hybrid language, presenting differences from the IEC 61131-3 standard. Such languages are common, as adaptations to specific domain PLCs may be necessary.

We projected a parser to analyze the programs; it deals with the elements of the standard languages and may be customized to specified differences, to accommodate new languages. This way we can deal with legacy programs that are not strictly standard compliant. To handle XML, we developed a reader module along with the default parser to load the PLC model.

Once the PLC model is constructed, we are able to work independently from the PLC programs to generate the B specification. It is also possible to generate a PLCopen XML, as documentation, to the PLCs that were not in this format.

3.2 Generation of the B Model

A good architecture is essential to generate a good model, as well as to define which information from the PLC model will be responsible for which elements of the B model. The architecture of this model is depicted in figure 3. It represents a POU *program* and the auxiliary POU *functions* or *function blocks* that it may

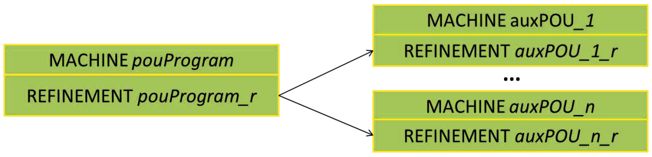


Fig. 3. B model representing a POU *program* and its use of auxiliary *function* and *function block* POUs

use; in the sense of B, they are included by the refinement of the component representing the POU *program*.

For the POU *program*[2], the **operations** are derived from the SFC steps. At the *machine* level, the bodies of the **operations** only make non-deterministic assignments to the variables modified in the respective step; the translation of the ST statements in the SFC action associated to the step forms the operation's body at the *refinement* level. The **precondition** of an **operation** is derived from the translation of the ST statements in the SFC transition preceding the respective step.

Variables are created to represent the internal representation of the POU inputs, named by the prefix "int_" plus the input's original name. These inputs are received as parameters in a **Start** operation, representing the *initialization* of the POU in each *execution cycle*. In the body of this operation, at the *refinement* level, each internal variable receives the value of its corresponding input.

The POU *outputs* are treated as local variables; it is no loss of generality to deal with them like that since we are dealing with the POUs only as independent components. The safety constraints will concern mostly these *outputs*.

To emulate the *execution cycle* of the POU, non-existent in B, a boolean variable is created for each step, named by the step's original name plus the suffix "_done". It is stated true as the respective **operation** is performed and falsified as the correspondent next **operation** in the *execution cycle* is reached. These variables will be part of the **operations**' preconditions: the predecessors step variables must be valid so that a step can be reached.

A variable *beginning* is also created, stated true in the **INITIALISATION** clause of the *refinement*, and is part of the precondition of the **Start** operation. It marks the first *execution cycle*, when **Start** must always be available. In its body *beginning* will be falsified.

In the *auxPOU_n* components, the **operations** are constructed with the translated statements from the auxiliary POUs, *functions* or *function blocks*, either in ST, IL, LD or FBD. In the POU *program*'s *machine* and *refinement* are created and typed variables according to the return type of these **operations'** outputs; they are used whenever one of them is invoked.

Further refinements may be performed to optimize the model, like adding invariants or changing its structure to facilitate automated proof.

[2] Due to space limitations and to the fact that our case study in this paper deals particulary with SFC and ST, this explanation covers only the elements of these languages.

3.3 Inserting the Safety Constraints

The next phase is to add safety constraints. Since PLC programs do not represent such constraints explicitly, they have to be manually extracted from the project requirements and modeled to be used in the formal models. This is a hard task and still an open issue in the industry [13], and we have not decided yet which methodology to adopt to tackle this problem. However, some promising approaches as [14] and [15] may be suited for our purpose; the latter was used in our case study.

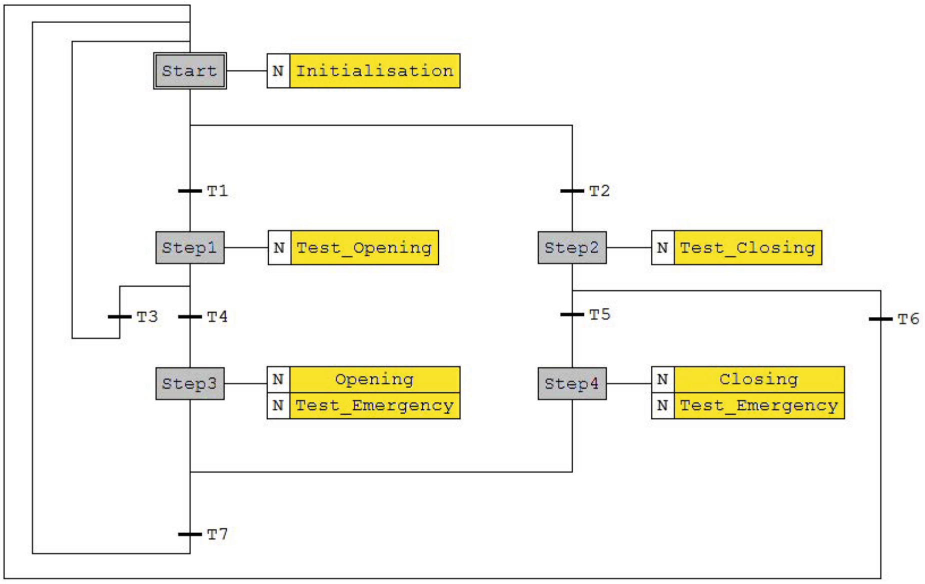


Fig. 4. SFC program for CDC. Execution goes from the initial step to sequential ones according to the validation of the transitions; the actions performed in each step are implemented in ST.

Once the safety constraints are defined, they are inserted into the model as **invariants** in the POU components, conditions that must always hold as the PLC actions are performed. Tools such as Atelier B[3] can perform automatic verification of their consistency and point out where lies any problem, guiding its treatment.

To guarantee that the PLC performs the expected behavior of its *execution cycle* we may create LTL formulas over the variables representing the **operations'** execution, verifying, e.g., if a given **operation** is ever reached from a predecessor one.

[3] http://www.atelierb.eu/

We may also verify properties that cannot be modeled with regular first order logic, requiring modalities found in LTL: one example is the condition that *whenever* a given variable has a given value, there must be *some* state reached by the PLC where another certain variable receives another certain value. This can be modeled with the operators $\Box$, meaning "it will *always* be the case that..."; and $\diamond$, meaning "it will *eventually* be the case that...".

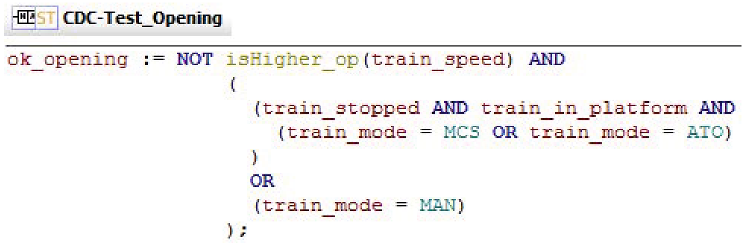
ProB is an animator and LTL model checker capable of handling B machines. It also provides support to verify structural properties such as *deadlock-freedom*.

4 Case Study

Our case study is the CDC (*Central Door Controller*), a PLC part of the doors subsystem of trains in the Metro-DF project, developed by AeS[4], a small company in Brazil specialized in railway projects.

The CDC is responsible for controlling the opening and closing of the doors in the train, guaranteeing that these actions are only executed under *safe* circumstances. It also controls the emergency situations that the train may be involved in, which must be taken in consideration to determine whether a given scenario is safe.

It receives, as input, information about the state of the train, such as the current speed, and commands to open or close the doors. After verifying if the conditions to execute some action are fulfilled, the CDC sends out commands, as outputs, allowing or not the required actions.



```
ok_opening := NOT isHigher_op(train_speed) AND
                  (
                    (train_stopped AND train_in_platform AND
                      (train_mode = MCS OR train_mode = ATO)
                    )
                    OR
                    (train_mode = MAN)
                  );
```

Fig. 5. ST action associated to the SFC program of CDC. Tests if the conditions to open the doors of the train are satisfied.

A simplified PLC representing the CDC is shown in figure 4, with a POU *program* in SFC, and its *transitions* and *actions* written in ST – they are not all presented due to space limitations, but in figure 5 we have the action *Test_Opening* in detail. The CDC interface is shown in table 1. Associated with the CDC is also a POU *function*, *isHigher_op*, which receives an integer as input and returns a boolean result indicating whether the input is higher than 6. This function is used to check the speed of the train.

[4] http://www.grupo-aes.com.br/site/home/

Table 1. Interface of CDC

Name	Class	Type
train_stopped	Input	BOOL
train_in_platform	Input	BOOL
train_speed	Input	INT
train_mode	Input	OPERATION_MODES
mech_emg_actuated	Input	BOOL
close_from_ATC	Input	BOOL
close_from_cabin	Input	BOOL
doors_closed	InOut	BOOL
ok_opening	Local	BOOL
ok_closing	Local	BOOL
emergency_evaluated	Local	BOOL
control_mech_emg_actuated	Output	BOOL
authorize_emergency	Output	BOOL
cab_emg_sound	Output	BOOL
interlock_doors_traction	Output	BOOL
apply_emg_breaks	Output	BOOL

– **Inputs**: environment state and commands.
– **Locals**: CDC operational variables.
– **Outputs**: Results of the CDC operations.

The execution begins at the step **Start**: the PLC reads the inputs of the external system and initialize its local variables. The transitions **T1** and **T2** will test if an opening or closing operation, respectively, was requested, then directing the execution to **Step1**, where the CDC tests if the conditions to *open* the doors are satisfied; or to **Step2**, where the CDC tests if the conditions to *close* the doors are satisfied. If the conditions either to open or to close the doors were satisfied, situation controlled respectively by the local variables *ok_opening* and *ok_closing*, the execution continues; otherwise it goes back to **Start**, where the CDC will wait until the next reading of inputs. In **Step3** or **Step4**, responsible respectively for opening and closing operations, the outputs controlling the state of the doors are modified, corresponding to the kind of action performed – opening or closing; the emergency circumstances are evaluated in these steps; the corresponding controlling outputs are also modified here.

4.1 Applying the Method

We use the tool Beremiz[5] to create the PLC program and obtain a PLCopen XML document representing the CDC. We translate the information in it into our PLC model, then generate a B model representing the CDC. The architecture of the generated model is presented in figure 6. This process is fully automatic.

The variables with the prefix "int_" are the internal variables created to represent the inputs received by the PLC. The local and the output variables are created with the same names as the ones shown in table 1. The auxiliary variable *aux_bool* is created to be used when the operation **isHigher_op** is invoked in *CDC_r*. The others are the step variables, used to represent the *execution cycle*, plus *beginning*, signaling the first execution.

In figure 7, we can see part of the B operation produced by the translation of **Start**. The precondition types the inputs and specify the conditions when **Start** can be executed: the first execution cycle – *beginning* = **TRUE**; the condition of transition **T3** is satisfied and the execution is in **Step1**; the condition of transition **T6** is satisfied and the execution is in **Step2**; or the condition of transition **T7** is satisfied and the execution is in **Step3** or **Step4**.

[5] `http://www.beremiz.org/`

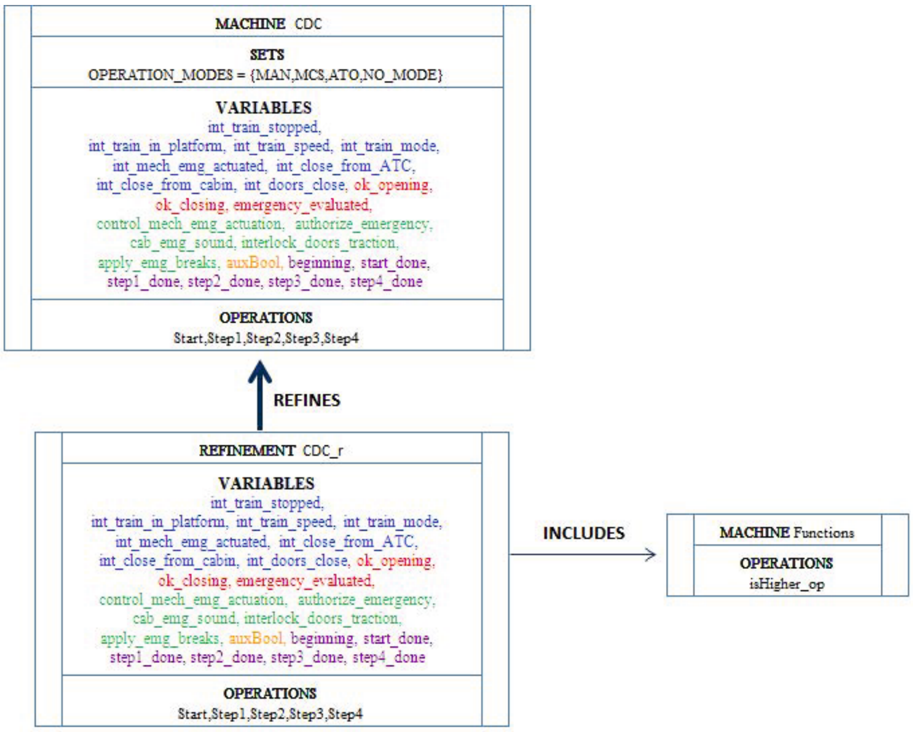

MACHINE CDC
SETS
OPERATION_MODES = {MAN,MCS,ATO,NO_MODE}
VARIABLES
int_train_stopped, int_train_in_platform, int_train_speed, int_train_mode, int_mech_emg_actuated, int_close_from_ATC, int_close_from_cabin, int_doors_close, ok_opening, ok_closing, emergency_evaluated, control_mech_emg_actuation, authorize_emergency, cab_emg_sound, interlock_doors_traction, apply_emg_breaks, auxBool, beginning, start_done, step1_done, step2_done, step3_done, step4_done
OPERATIONS
Start,Step1,Step2,Step3,Step4
REFINES
REFINEMENT CDC_r
VARIABLES
int_train_stopped, int_train_in_platform, int_train_speed, int_train_mode, int_mech_emg_actuated, int_close_from_ATC, int_close_from_cabin, int_doors_close, ok_opening, ok_closing, emergency_evaluated, control_mech_emg_actuation, authorize_emergency, cab_emg_sound, interlock_doors_traction, apply_emg_breaks, auxBool, beginning, start_done, step1_done, step2_done, step3_done, step4_done
OPERATIONS
Start,Step1,Step2,Step3,Step4
INCLUDES
MACHINE Functions
OPERATIONS
isHigher_op

Fig. 6. Architecture of the B model generated representing the CDC

$Start(train\_stopped, train\_in\_platform, train\_speed, train\_mode, mech\_emg\_actuated,$
$close\_from\_ATC, close\_from\_cabin, doors\_closed) =$

PRE

$train\_stopped : \mathbf{BOOL}$ & $train\_in\_platform : \mathbf{BOOL}$ & $train\_speed : \{0, 5, 10\}$
& $mech\_emg\_actuated : \mathbf{BOOL}$ & $close\_from\_ATC : \mathbf{BOOL}$ &
$close\_from\_cabin : \mathbf{BOOL}$ & $train\_mode : OPERATION\_MODES$ &
$doors\_closed : \mathbf{BOOL}$ & $(beginning = \mathbf{TRUE}$ **or** $((\mathbf{not}(ok\_opening = \mathbf{TRUE})$
& $step1\_done = \mathbf{TRUE})$ **or** $(\mathbf{not}(ok\_closing = \mathbf{TRUE})$ & $step2\_done = \mathbf{TRUE})$
or $(emergency\_evaluated = \mathbf{TRUE}$ & $(step3\_done = \mathbf{TRUE}$ **or**
$step4\_done = \mathbf{TRUE}))))$

(...)

Fig. 7. Start operation in the CDC *machine*. Only the precondition is exhibited.

The body of the **Start** operation at the *refinement* level, shown in 8, consists of the translation of the statements in the ST action—the initialization of the local variables—, plus the generated initialization of the variables representing the outputs; then the assignments of the inputs to its internal variables; and finally the initialization of the step variables, marking the active step as **Start** – $start\_done := \mathbf{TRUE}$.

```
$Start(train\_stopped, train\_in\_platform, train\_speed, train\_mode, mech\_emg\_actuated,$
$close\_from\_ATC, close\_from\_cabin, doors\_closed) =$
  BEGIN
    $ok\_opening := $ FALSE; $ok\_closing := $ TRUE; $emergency\_evaluated := $ FALSE;

    $control\_mech\_emg\_actuation := $ FALSE; $authorize\_emergency := $ FALSE;
    $cab\_emg\_sound := $ FALSE; $interlock\_doors\_traction := $ FALSE;
    $apply\_emg\_breaks := $ FALSE;

    $int\_train\_stopped := train\_stopped$; $int\_train\_in\_platform := train\_in\_platform$;
    $int\_train\_speed := train\_speed$; $int\_mech\_emg\_actuated := mech\_emg\_actuated$;
    $int\_close\_from\_ATC := close\_from\_ATC$;
    $int\_close\_from\_cabin := close\_from\_cabin$;
    $beginning := $ FALSE; $start\_done := $ TRUE; $step1\_done := $ FALSE;
    $step2\_done := $ FALSE; $step3\_done := $ FALSE; $step4\_done := $ FALSE
  END
```

Fig. 8. Start operation in the CDC_r *refinement*

```
$Step1 =$
  PRE
    not($int\_close\_from\_ATC = $ TRUE or $int\_close\_from\_cabin = $ TRUE)$^{\mathbf{T1}}$
    & $start\_done = $ TRUE
  THEN
    $start\_done := $ FALSE; $auxBool$ <-- $isHigher\_op(int\_train\_speed)$;
    IF $auxBool = $ TRUE THEN
      /*block opening*/
      $ok\_opening := $ FALSE
    ELSE IF (($int\_train\_mode = $ MAN) or (($int\_train\_mode = $ MCS or
       $int\_train\_mode = $ ATO) & ($int\_train\_stopped = $ TRUE)
       & ($int\_train\_in\_platform = $ TRUE)))
      THEN
        /*Opening allowed*/
        $ok\_opening := $ TRUE
      ELSE
        /*block opening*/
        $ok\_opening := $ FALSE
      END
    END;
    $step1\_done := $ TRUE
  END
```

Fig. 9. Operation representing **Step1** (The precondition of the *machine* operation is exhibited together with the *refinement* operation due to space limitations

The B operation resulting from the translation of **Step1** is shown in figure 9. In its precondition we have **T1** and the obligation that **Step1**'s predecessor step

```
INVARIANT
  ( (ok_opening = TRUE) =>
     ( (int_train_speed<=6) & ( ( (int_train_mode=MCS or int_train_mode=ATO)
      & (int_train_stopped = TRUE) & (int_train_in_platform = TRUE) )
       or (int_train_mode = MAN) )
     )
  ) &
  ( (ok_closing = FALSE) =>
      (int_train_mode = ATO & int_close_from_cabin = TRUE)
  )
```

Fig. 10. Invariants concerning opening and closing safety

variable, *start_done*, must be valid. Its body statements are the translation of the ST statements in the step's associated action, *Test_Opening*, shown in figure 5.

The other operations are generated according to the same guidelines. Once the B model is ready, the next phase in our method is to insert, manually, the *safety constraints* of the project as *invariants*. We used the ProR approach [15] to define the formal constraints from the natural language requirements, easing the process and assuring reliable traceability; the whole effort is in [16]. We present here only the results to the following requirements, concerning the opening and closing operations:

1. The doors shall open only when the train's speed is lower than or equal to 6km/h.
2. The conditions to open all the doors located in one or in the other side of the train, when in the operation mode ATO or MCS, are the train be stopped and in the platform.
3. The condition to open all the doors located in one or in the other side of the train, when in the operation mode MAN, is the train's speed be lower than or equal to 6km/h.
4. In ATO mode, the Central Door Controller must not close the doors while receiving the command to open them from the driver push buttons.

We have the resulting B invariants in figure 10. The first invariant, referring to the situations where opening is *allowed*, covers the items 1, 2 and 3. The second invariant, referring to the situation where closing is *prohibited*, covers the item 4. The model is then ready to formally verify them.

The last phase of the process is to create the LTL formulas to check if the PLC program's behavior is as expected. Concerning the *execution cycle*, the conditions to be verified and the respective formulas are shown below:

1: **Start** is reachable :	1: $\diamond\ start\_done$
2: **Step2** must be reachable from **Start** when **Step1** is not :	2: $\Box((start\_done \wedge \neg \diamond\ step1\_done) \Rightarrow\ \diamond step2\_done)$
3: **Step1** must be reachable from **Start** when **Step2** is not :	3: $\Box((start\_done \wedge \neg \diamond\ step2\_done) \Rightarrow\ \diamond step1\_done)$
4: **Step3** must be reachable from **Step1** when **Start** is not :	4: $\Box((step1\_done \wedge \neg \diamond\ start\_done) \Rightarrow\ \diamond step3\_done)$
5: **Start** must be reachable from **Step1** when **Step3** is not :	5: $\Box((step1\_done \wedge \neg \diamond\ step3\_done) \Rightarrow\ \diamond start\_done)$
6: **Step4** must be reachable from **Step2** when **Start** is not :	6: $\Box((step2\_done \wedge \neg \diamond\ start\_done) \Rightarrow\ \diamond step4\_done)$
7: **Start** must be reachable from **Step2** when **Step4** is not :	7: $\Box((step2\_done \wedge \neg \diamond\ step4\_done) \Rightarrow\ \diamond start\_done)$
8: **Start** must be reachable from **Step3** or **Step4** :	8: $\Box((step3\_done\ \vee\ step4\_done) \Rightarrow\ \diamond start\_done)$

We also verify constraints non-expressible through the invariants, such as:

9: Always when the CDC attests that the conditions to open are satisfied, then the doors must open at some point of its execution :	9: $\Box(ok\_opening\ \Rightarrow \diamond \neg int\_doors\_closed)$
10: Always when the CDC attests that the conditions to close are satisfied, then the doors must close at some point of its execution :	10: $\Box(ok\_closing\ \Rightarrow \diamond\ int\_doors\_closed)$

4.2 Results

Once the model is complemented with the invariants representing the safety constraints and the LTL formulas to verify the program's behavior are defined, we are in position to carry on formal verification through theorem proving of proof obligations, model checking and LTL formulas check.

Ten proof obligations were generated to verify the invariants inserted in the model: 6 in the operation **Step1**, related to the invariant concerning the opening conditions; and 3 in the operation **Step4**, associated with the invariants representing the emergency conditions, not exhibited here due to space limitations. The Atelier B theorem prover was able to prove them all automatically, without any user interaction. The operation **Step2** does not generate proof obligations because its statements are strictly equal to the invariant concerning the closing conditions, and the operation **Step3**, as it opens the doors, directly satisfies all the invariants concerning emergency conditions, by vacuity.

Next, we model check the model for the properties not covered by the proof obligations, such as *deadlock-freedom* and *liveness*. As a result we had 4969 states, all free from deadlock, and a total of 1792080 transitions were necessary to cover them all. An important observation is that in order to avoid a state-explosion problem we restricted, *only* for the model checking phase, the values of the **INT** variables - *train_speed* and *int_train_speed* - to $\{0, 5, 10\}$; there is no loss of generality, since the chosen values represent the three possible states of the train: "*stopped*", "*in movement and with its speed lower than 6km/h*" and "*in movement and with its speed not lower than 6km/h*". Without this restriction an infinite number of states would have be generated by the model checker to cover the possible values of the **INT** variables.

We can see in table 2 that most of the operations performed in the model checking phase were at **Start**, where the inputs are received and the *execution cycle* of the PLC is initiated. The computing time was of ten minutes.

Table 2. Total of transitions covered

Operation	Number of visits
Initialisation	1536
Start	1787904
Step1	384
Step2	1152
Step3	48
Step4	1056

The final step was to check the LTL formulas to verify if the PLC was presenting the expected behavior. All the formulas were proven correct, so the CDC was indeed executing as planned. The computing time was of less than one minute per formula.

5 Conclusions and Future Work

We have overviewed a method to carry out formal verification of PLC programs, according to the IEC 61131-3 standard, through the automatic generation of a B specification. Safety constraints are inserted in the formal model and then verified through theorem proving; we also verify structural properties and if the PLC presents the expected behavior performing model checking and using LTL formulas. Thus we increase the reliability of the application, having correctness according to the specification as a realistic and achievable goal.

Another key point of our approach is that, as it allows the users to generate the B models automatically from the PLC programs, only lacking the safety properties, it boosts the process of formal verification of such programs, skipping all the hard work to design and construct the model that prevents formal methods from being easily inserted in industrial projects.

We also presented a case study in a real railway application where our approach was applied with success. We were able to attest the efficacy of the automatic provers and verify the safety constraints of the project.

Future work lies mostly in expanding and adjusting the generation of the B models, improving the way the method deals with some issues, such as multi-dimensional arrays, loops and data types supported, for instance. Scalability is also an issue, since the bigger and more complex the generated models are, the harder it is to verify them; we plan to tackle this exploiting the decomposition support of B, splitting the complexity of the application in several components and verifying them independently.

The results obtained from the model and LTL checking can be used to improve the PLC, but we have not defined yet an appropriate methodology on how to perform this improvement; that is in our future works as well. We are also studying the automatization of the process of deriving the safety constraints from the requirements.

To improve the confidence in our translation method, another future work is to make the inverse process: generate the former PLC programs from the B models, so that we can apply testing technique to validate our approach.

As we expand the scope of our method, we also intend to perform more case studies. We are about to start one with the company ClearSy[6], strongly involved with the B method and safety critical systems engineering, in a real project also in the railway field, to execute problem diagnosis in high speed trains.

References

1. IEC (2003): IEC 61131-3 - Programmable controllers. International Electrotechnical Comission Standards (2003)
2. Kron, H.: On the evaluation of risk acceptance principles. In: 19th Dresden Conference on Traffic and Transportation Science (2003)
3. Amey, P.: Dear sir, yours faithfully: an everyday story of formality. IN Proc. 12th Safety-Critical Systems Symposium, p. 318 (2004)
4. Parnas, D.: Really rethinking 'formal methods'. Computer (January 2010), `http://portal.acm.org/citation.cfm?id=1724964.1724987`
5. Ljungkrantz, O., Åkesson, K., Fabian, M., Yuan, C.: A Formal Specification language for PLC-based Control Logic. In: Proc. of 8th IEEE International Conference on Industrial Informatics, pp. 1067–1072 (2010)
6. Soliman, D., Frey, G.: Verification and Validation of Safety Applications based on PLcopen Safety Function Blocks using Timed Automata in Uppaal. In: Proceedings of the Second IDAC Workshop on Dependable Control of Discrete Systems (DCDS), pp. 39–44 (2009)
7. Farines, J., de Queiroz, M.H., da Rocha, V.G., Carpes, A.A.M., Vernadat, F., Crégut, X.: A model-driven engineering approach to formal verification of PLC programs. In: IEEE EFTA (2011)
8. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, Cambridge (2005)
9. PLCopen : XML Formats for IEC 61131-3. PLCopen Technical Committee, 6 (2009)
10. Barbosa, H., Déharbe, D.: Towards formal verification of PLC programs. In: 14th Brazilian Symposium on Formal Methods: Short Papers, São Paulo- SP (2011)

[6] `http://www.clearsy.com/`

11. Barbosa, H., Déharbe, D.: Formal Verification of PLC Programs Using the B Method. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 353–356. Springer, Heidelberg (2012)
12. Lecomte, T., Servat, T., Pouzancre, G.: Formal methods in safety-critical railway systems. In: Proc. Brazilian Symposium on Formal Methods: SMBF (January 2007)
13. Abrial, J.R.: Formal methods in industry: achievements, problems, future. In: Proceedings of the 28th International Conference on Software Engineering, pp. 761–768 (2006)
14. Cabral, G., Sampaio, A.: Formal Specification Generation from Requirement Documents. In: SBMF (2006)
15. Ladenberger, L., Jastram, M.: Requirements Traceability between Textual Requirements and Formal Models Using ProR
16. Barbosa, H.: Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o método B. B.Sc. Thesis, UFRN, DIMAp, Natal, Brazil (2010)