processes

MDPI

Article

Fresh Approaches for Structured Text Programmable Logic Controllers Programs Verification

Émile Siboulet [1], Louen Pottier [1], Tom Ranger [2] and Bernard Riera [2,*]

[1] Département de Génie Mécanique, École Normale Supérieure Paris-Saclay, 91190 Gif-sur-Yvette, France
[2] CReSTIC, Université de Reims Champagne-Ardenne, 51100 Reims, France
* Correspondence: bernard.riera@univ-reims.fr

Abstract: Programmable logic controllers (PLCs) are everywhere today and perform critical tasks in industries. They are considered as a key component for the Industry 4.0. Before they are put into operation, it is necessary to check the accuracy of the PLC programs. This verification operation can be performed using model checkers. This stage is often long and costly and requires a domain expert who can understand the system, as well as the different model checker tools able to verify the code implemented in the controller. Furthermore, this verification often requires a conversion of the PLC code into a language understood by a model checker which can influence the behavior of the observed PLC. Hence, there is a need to propose methods and tools which could be used by technicians and engineers. The aim of this paper is to propose methods that require little work to set up and are robust to program sizes used in Industry 4.0. This paper explores some fresh ideas for human-adapted PLC code verification. We present different methods to test codes in structured text (ST) compliant with the IEC 61131-3 standard. Hence, the first idea is to test the ST code that will be directly implemented on a controller. For that, we propose a method using the model checker UPPAAL which allows us to obtain exact results on short codes. Second, we propose verifying the generic properties that a PLC program must avoid: deadlocks, non-accessible states and fugitive states or actions. To solve combinatory explosion problems encountered with the UPPAAL software, the third proposition consists of using relational databases. The same verification as previously followed can be obtained, but the search time is longer. The fourth and last proposal is to process the ST code with a neural network composed of long short-term memory layers (LSTM) to quickly determine the validity of the code. This method could give an approximation of code errors in a few seconds. The different proposed methods are supported with several examples.

Keywords: system verification; model checking; relational databases; recurrent neural networks; programmable logic controllers; automation; structured text

Citation: Siboulet , É.; Pottier, L.; Ranger, T.; Riera, B. Fresh Approaches for Structured Text Programmable Logic Controllers Programs Verification. *Processes* **2023**, *11*, 687. https://doi.org/10.3390/10.3390/pr11030687

Academic Editor: Anet Režek Jambrak

Received: 31 January 2023
Revised: 18 February 2023
Accepted: 21 February 2023
Published: 24 February 2023

1. Introduction

Industry has entered a phase of big change that sees digital technologies as a key factor for the future to design cyber–physical production systems. The convergence of the virtual world of the Internet and information technology (IT) and the real world of industrial installations and operational technology (OT) will be the challenge for the Factory of the Future (Industry 4.0). Modern information and communication technologies seem to be a solution to increase productivity, quality and flexibility within industry. The challenges are numerous: connectivity and interoperability, digital twins, decentralization, human-centered automation... These systems are predicted to enable new automation paradigms and improve plant operations in terms of increased facility effectiveness. However, that could be accomplished only if innovative methodologies and tools are given to the control engineer. One of them concerns programmable logic controllers (PLCs), program verification tools. PLCs, with their sequential and cyclic operation, specified by the IEC 61131 standards are widely used today for the control of automated installations [1].

Article [2] shows that the programming model for PLCs needs to evolve to support the increasing complexity and more extensive network integration demanded by Industry 4.0, but PLCs will remain key components in Industry 4.0. Furthermore, refs. [3–5] give a deep description of risk management in a factory where PLCs work together. In previous works, we have proposed original methodological approaches to bridge the gap between formal approaches in academia and industrial applications by proposing an original workflow for automation study [6,7]. This workflow is based on an automatic generation process of deliverables (PLC programs, recipe book...). However, the problem of PLC code verification remains a scientific and technical issue. Our work in this paper is thus complementary to the abovementioned approaches. The validation stage is generally based on factory acceptance testing (FAT) carried out in the supplier's test facilities and site acceptance testing (SAT) carried out at the customer's site. The commissioning of controllers remains a critical step today and for Industry 4.0, as it is time-consuming and stressful for the staff. Correcting errors at this stage of the project is particularly costly. Verification and "virtual commissioning" of installations are now scientific and methodological obstacles to Industry 4.0. Today, we note that automation engineers use few or no code verification tools from the academic world, formal or not, to facilitate the validation and commissioning stage [8]. The reason these practices are not used in the industrial world is mainly related to their usability. They are considered too complex, given the models, formalism and methods used, and often disconnected from the real needs of automation specialists.

Our goal is to propose a set of methods that allow checking some properties common to each correctly programmed PLC code during development and prior to the implementation of the facility. In order to automatically find faults before implementation, we must restrict ourselves to indicating the presence of certain errors that will almost certainly induce problems during code execution. Errors will be found on the variables that compose the code to avoid the demanding task of code formalisation . We will focus on extracting accessible states, non-accessible states, deadlock, and fugitive states from the structured text (ST), also defined in IEC 61131-3 and implemented on PLCs. Usually, a PLC program is a cyclic sequence. The program includes a few variables that are in deadlock, which makes its systematic search relevant. It is the same for fugitive states; we expect a PLC to react to situations. A fugitive state is a change in the parameters of the PLC without any external event that can involve fugitive actions which, most of the time, is a problem. From the accessible and non-accessible states, a competition matrix can be produced which can give more information about the code. In order to explain the relevance of the methods developed, we will apply our methods to different examples to test the limits of considered approaches.

Software exists that can perform a formal proof of the code. Examples include UPPAAL [9], NuSMV [10] or Supremica [11]. However, it is necessary to convert the code implemented on the PLC to have an overview of the reliability of the implemented code as noted in [12–14]. To remove this source of error, we propose a method similar to [15] or [16] which involves verifying properties from a specification. To our best of knowledge, no other study examines this specific application context. This method is much more computationally expensive. The error search performed by UPPAAL, as noted in [17] keeps the information necessary for its completion in RAM. The amount of information sufficient to ensure the reliability of a code can be very important; it is necessary to find an effective method for storage and research of the information once generated. To solve this problem, we propose organizing the information in a database by converting the logical code into a relational database similar to [18–21]. This allows optimizing the storage space to allow fast retrieval of information. We also propose a search for errors in the PLC code by queries in the database. Thus, the error search is no longer an exploration of a graph in a PLC but by a query on a database. This method is particularly time-consuming because the information is stored on a computer's hard drive. Nevertheless, this search for errors can be distributed among several computers by following the method detailed by [22] which uses fast communication between FPGA (field-programmable gate array). However, this

precise study of complex code is often not necessary during code development, but it is convenient to quickly obtain information. Text understanding by using neural networks is a fast-growing problem. Although very standardized, this operation is not simple. Indeed, unlike functions to be reproduced or images to be analyzed, all the texts we want to analyze are not the same size, and there is no simple way to force a text into a precise size unlike an image. It is thus necessary to analyze a text as a sequence of words; for that we can use the recurrent neural networks as detailed in [23,24]. Neural networks composed of layers of long short-term memory (LSTM), as noted in [25], are ideal for this task as shown by [26]. It is natural to try to find properties in computer program codes with LSTMs, as noted in [27,28]. By analyzing the code with a recurrent neural network, we can obtain results similar to the formal methods.

The rest of this paper is organized as follows. Section 2 examines the framework of the methods we will use as well as some notations in codes used as examples. Section 3 shows how it is possible from the ST code of a PLC to generate a graph in the UPPAAL format. This method is efficient, but only a small verification can be carried out because of the research without the storage of UPPAAL. This issue is addressed in Section 4, where we explain how a database can be used to store and retrieve information to check larger examples. Section 5 explains how to go even further by using recursive neural networks to sequence the code to find errors without exploring the marking graph. The conclusion of the paper is given in Section 6.

2. Scope of Use

2.1. Illustrative Example

As explained in the introduction, one of the main challenges of using automatic verification is for the user to define the functional and safety properties that have to be checked. These are strongly dependent on the industrial process [29] and require strong and specific competencies to express them in a formal language. In this paper, one of the ideas is to focus on properties that are important in all PLC programs for automation engineers. Hence, the use of the verification tools does not require a model-checker expert. From our experience [6,7], we have selected four main errors to avoid in PLC programs: deadlocks, accessible or non-accessible states, and fugitive states or actions. The purpose of the following lamps example is to show the errors that different proposed methods must detect. Figure 1 shows the specification of a PLC with 3 lamps: $L1$, $L2$, $L3$, driven respectively by the 3 inputs: A,B,C. $L1$ needs a rising edge to make the switch between its states. This control is correct. $L2$ changes its state when it is triggered. The absence of the rising edge will cause the PLC to not stop on the next state. This example will show a fugitive state and action when B is triggered. $L3$ does not allow a return to the initial state and therefore cannot turn off; we must then obtain a deadlock. The ST code will be obtained from the sequential function chart (Grafcet) automatically, thus ensuring the good functioning of the code [30].

The previous example is small. We choose an example of a heavier operative part which cannot be certified by the current methods depicted in Figure 2. This figure was obtained with the software *Factory I/O* which allowed us to test the codes used in an experiment simulating the behavior of the sorting station considered. This example is a separating station which sorts boxes of two different colors. It consists of four conveyor belts, two mono-stable cylinders, six Boolean sensors and two colors sensors that differentiate the boxes. The controller code of this unit is 250 lines long and is composed of 183 internal variables. This code is also obtained from the automatic conversion of a grafcet [30] that has been thoroughly checked by several automaticians. No formal verification has been able to check the accuracy of this code apart from the use of the method using a database written in Section 3. Thus, we make the assumption of its validation which will have been tested by simulation and virtual commissioning.

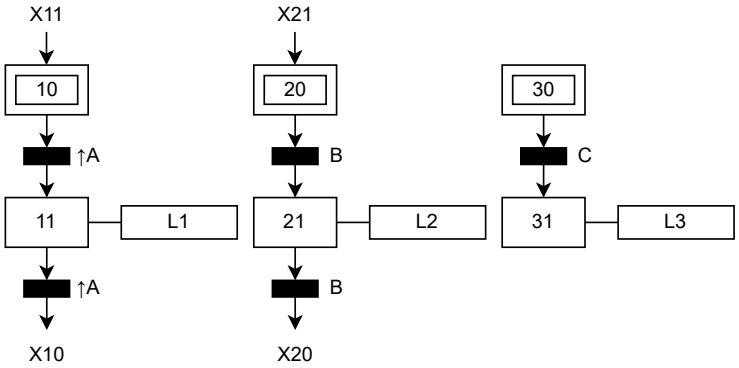


Figure 1. Grafcet of the operative part of the lamps example.

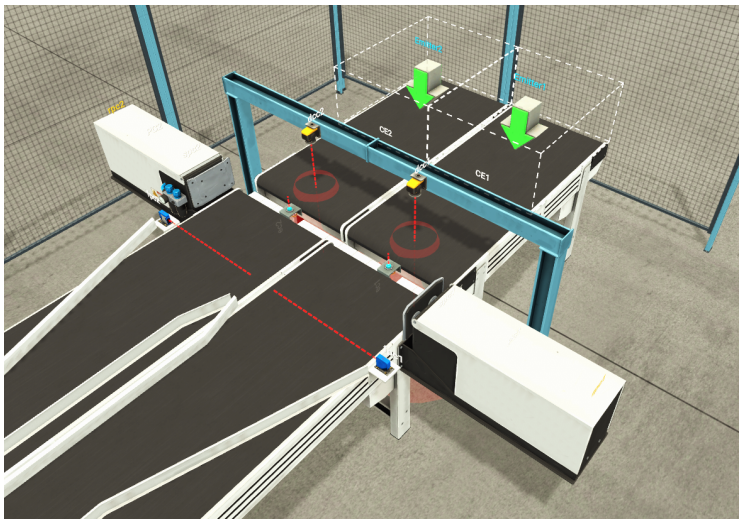


Figure 2. Separating station used as the heavy example.

2.2. Classes of Variables

Different classes of variables are distinguished and require different treatment in the modeling of the controller behavior. From the Grafcet shown in Figure 1, a code is automatically generated (Figure 3). We distinguish five types of variables:

- Inputs (red) which cannot be controlled and need to be generated externally to test the code (sensors for instance);
- Outputs (purple) which are the output of the controller where marking sequences need to be tested (actuators for instance);
- Internal variables (yellow) which are needed to establish the state of the controller at any given time. In the example, step variables ($X10$, $X11$, $X20$, $X21$, $X30$, $X31$) and the memories to calculate rising edge ($RE0p$, $RE1p$) are considered as internal variables;
- Recalculated internal variables (green) which are not needed to establish the state of the controller at any given time. Transition function variables ($FTX10$, $FTX11$, $FTX20$, $FTX21$, $FTX30$) and memory of input variables ($RE0$, $RE1$) in the example are considered as recalculated internal variables;
- Variables of initialization (blue) whose values are initialized to one during the start of the PLC and then switch to zero during operation.

These distinctions will be useful during the automatic verification stage.

```
RE0 := A;
FTX10 := X10 AND RE0 AND NOT RE0p;
RE1 := A;
FTX11 := X11 AND RE1 AND NOT RE1p;
FTX20 := X20 AND B
FTX21 := X21 AND B;
FTX30 := X30 AND C;
X10 := FTX11 OR X10 AND NOT FTX10 OR init;
X11 := (FTX10 OR X11 AND NOT FTX11) AND NOT init;
X20 := FTX21 OR X20 AND NOT FTX20 OR init;
X21 := (FTX20 OR X21 AND NOT FTX21) AND NOT init;
X30 := X30 AND NOT FTX30 OR init;
X31 := (FTX30 OR X31) AND NOT init;
L1 := X11;
L2 := X21;
L3 := X31;
RE0p := RE0;
RE1p := RE1;
```

Figure 3. Colorized code by class of variables.

2.3. Code Restrictions

The structured text syntax allows edges on variable values. This syntax is specific to the PLC and will be replaced in the controller code by an internal variable storing the value at the previous PLC cycle. To ensure the end of the cycle of the PLC, the while loop is forbidden. It could be replaced by a for loop with a break statement on a condition. Timed triggers are considered as Boolean input variables in the same way as sensors. As timed triggers can occur at any time during operation, more coercive models could hide errors.

3. Automatic Verification Based on UPPAAL

3.1. Principle

Based on the code and the class of the variable, we need to create states graphs to simulate the behavior of the PLC. The circles represent states. They are connected to each other by transitions represented by arrows that are crossed where a synchronization happened which is marked after the semicolon. The exclamation mark represents the emission of the synchronization; the question mark represents the waiting of the synchronization. At the time of the transition between two states, the code marked before the semicolon is executed. For each type of variable previously established, we generate a graph that reflects the behavior of a PLC. For each step, any change can occur on any input variables. The code from the PLC is stored in a function created by the user called $grafcet()$, and values of variables are stored by UPPAAL. To explore the accessible marking graph, we use a synchronization graph that changes the input variables and executes the code (Figure 4).

The synchronization *var*! allows another graph to change variable values. Before the main loop, an initialization is conducted to start the PLC with the correct values. In a cycle, the PLC code is executed three times without any input change to detect fugitive states. To make the values of the input variables change, we use graphs similar to the graph in Figure 5a adapted to the possible values of this variable. Each input variable needs a graph.

With this method, fugitive states are tested at the end of the $grafcet()$ call by comparing the current value of the variable with its previous and pre-previous value. Deadlocks can be automatically tested by looking for the possibility of falling edges for each cycle. Accessible marking and non-accessible marking can be obtained with the UPPAAL query syntax. The synchronization of its graphs allows testing the Cartesian product of the inputs for each state of the code. To simplify the number of test inputs, we can impose restrictions on it. We can take the example of the cylinder end of stroke sensors that cannot both be triggered at the same time. This case does not need to be tested and allows us to greatly reduce the search space. This can be implemented by adding a non-simultaneous inputs graph (Figure 5b).

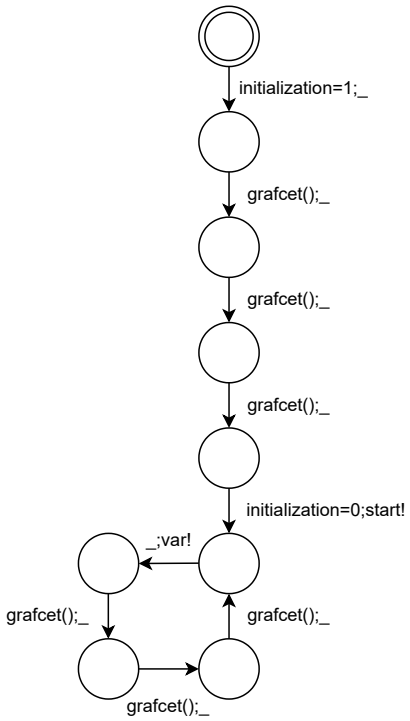
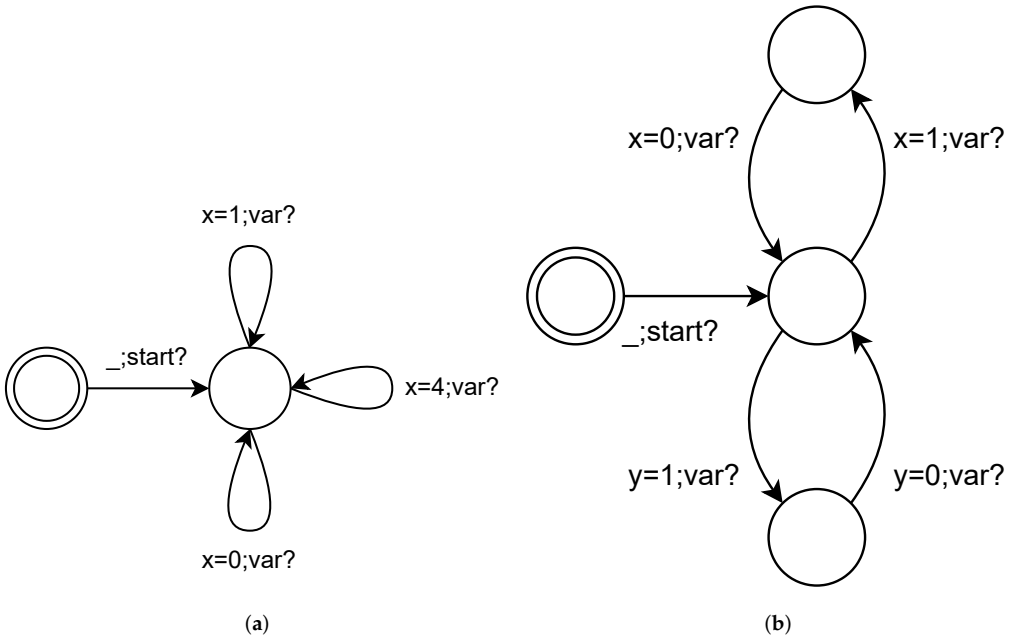


Figure 4. UPPAAL graph of synchronization.



(**a**) (**b**)

Figure 5. UPPAAL variables graph: (**a**) variables modification graph; (**b**) non-simultaneous inputs graph.

To generate these different graphs, we only needed:

- The ST code to be implemented on the PLC;
- The computer type of each variable and their possible values;
- The variable type defined in Section 2.2.

The ST code is transcribed in UPPAAL by changing the operators and graphs that are generated and instanced according to the rescued variables. To implement non-

simultaneous inputs, a list of non-simultaneous sensors is given to the simulation. The robustness of the code to sensor failures cannot be assessed anymore.

3.2. Results

To evaluate the fugitivity of a variable, we add lines of code to the PLC. These lines allow us to store the values of the variables during the PLC cycles. Thus, we define storage variables $L1p$ and $L1pp$ containing the values of $L1$ at the previous cycles. Then, we obtain the variable $L1f$ which contains the information of the fugitivity obtained by the allocation Figure 6. Variables that evaluate fugitivity determine the shape of the synchronization graph (Figure 4), which performs three PLC cycles before testing the different variables. UPPAAL queries are automatically generated and show correct behavior (Table 1). This result is consistent with the Grafcet specification.

```
L1f:= (L1pp and not L1p and L1) or (not L1pp and L1p and not L1);
```

Figure 6. Declaration of the variables that check fugitivity.

Table 1. Results of the lamps example by automatic verification using UPPAAL.

	L1	L2	L3
There is a state where the variable is fugitive.	False	True	False
There is a state where the variable is in deadlock.	False	Flase	True

3.3. Discussion

This method allows us to quickly obtain results on a large part of the code used on PLCs. Indeed, no restriction in the code is required. Moreover, the PLC is implemented in UPPAAL and can be submitted for other tests if one knows the syntax of the requests. However, we are very limited by the size of the PLC code. Indeed, UPPAAL stores the verification information in the RAM of the computer. In the case of large PLCs, it is not possible to contain all this information. In our case, it was not possible to use this method on the separating station. To solve this problem, it is necessary to find an algorithm allowing storing the graph of the accessible markings and which would be able to recover the information necessary at the validation of the code.

4. Exploitation of the Markings Graph by a Database

4.1. Principle

Keeping the graph of the markings in a database could save a lot of time if it has an adapted shape. The generation will be longer because of the storing time, but after this step, the query will be faster. The ST code is used as previously and is tested for each state with all possible inputs. The PLC simulator gives us for each founded state, starting from the initial state, new states with each possible input after a cycle as described in [17]. The retrieve data can be stored in a structured query language (SQL) database format as represented in Figure 7. To fill this database, we start with the nodes, the simplified nodes, the edges and the inputs marking which are directly obtained from the PLC simulator. The simplified nodes are the reduction of the marking of internal variables and output variables to output variables. This feature allows a faster search of the accessible output marking while keeping the total information of the microcontroller during the code correction phase. Fugitive states are tested afterwards via an SQL query (Figure 8) to ensure maximal efficiency during database population. Used table names in the query are the same as those in the graph representing the different relationships in Figure 7. This request is the most demanding for the software; in some cases, it is impossible to evaluate. Other results can be accessed by this method, such as accessible marking state and non-accessible marking state, by using recurrent requests on the nodes names.

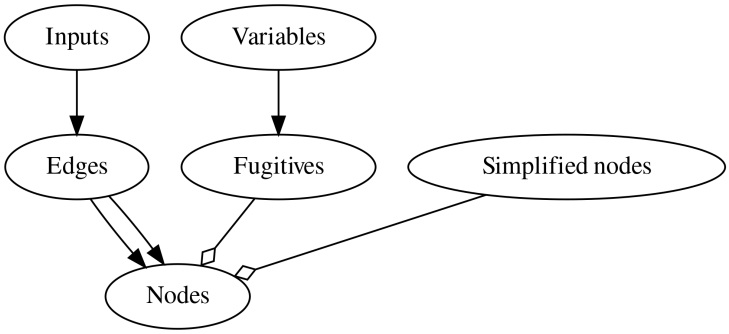


Figure 7. Shape of the database that stores marking graph.

```
SELECT target.marking, nd.marking, source.marking, nd.id
FROM Nodes AS nd
JOIN Edges AS T1 ON T1.target_node_id = nd.id
JOIN Edges AS T2 ON T2.target_node_id = nd.id
JOIN Nodes AS target ON T2.target_node_id = target.id
JOIN Nodes AS sources ON T1.source_node_id = source.id
WHERE T1.id = T2.id
```

Figure 8. SQL request to find a fugitive state.

4.2. Results

With the method proposed in this section, we can study the graph of the reachable markings in the example of the lamps represented in Figure 9. Red arrows represent transition between two strongly connected parts of a graph which indicates the presence of deadlocks. Green arrows come from a fugitive state and point to the fugitive output. We obtain similar results to the verification carried out with UPPAAL. On the example of the separating station, we have access to the concurrency matrix (Table 2) that shows $PC1$ and $PC2$ (two plunger actuators who share working space) cannot be activated at the same time. Moreover, $PC1$ and $PC2$ which share the working space with $C1$ and $C2$ (two conveyor belts) must not be activated at the same time. We see that this condition is also respected. Finally, $CE1$ and $CE2$ (two conveyor belts) do not share any work space and could be activated at any time. The generated database stores sixty million edges, twenty-six thousand states and weight height gigabytes. Fugitive states could not be computed. The database generation required twelve hours on a working station.

Table 2. Concurrency matrix obtained by the database.

	C1	**C2**	**CE1**	**CE2**	**PC1**	**PC2**
C1	1	1	1	1	0	0
C2	1	1	1	1	0	0
CE1	1	1	1	1	1	1
CE2	1	1	1	1	1	1
PC1	0	0	1	1	1	0
PC2	0	0	1	1	0	1

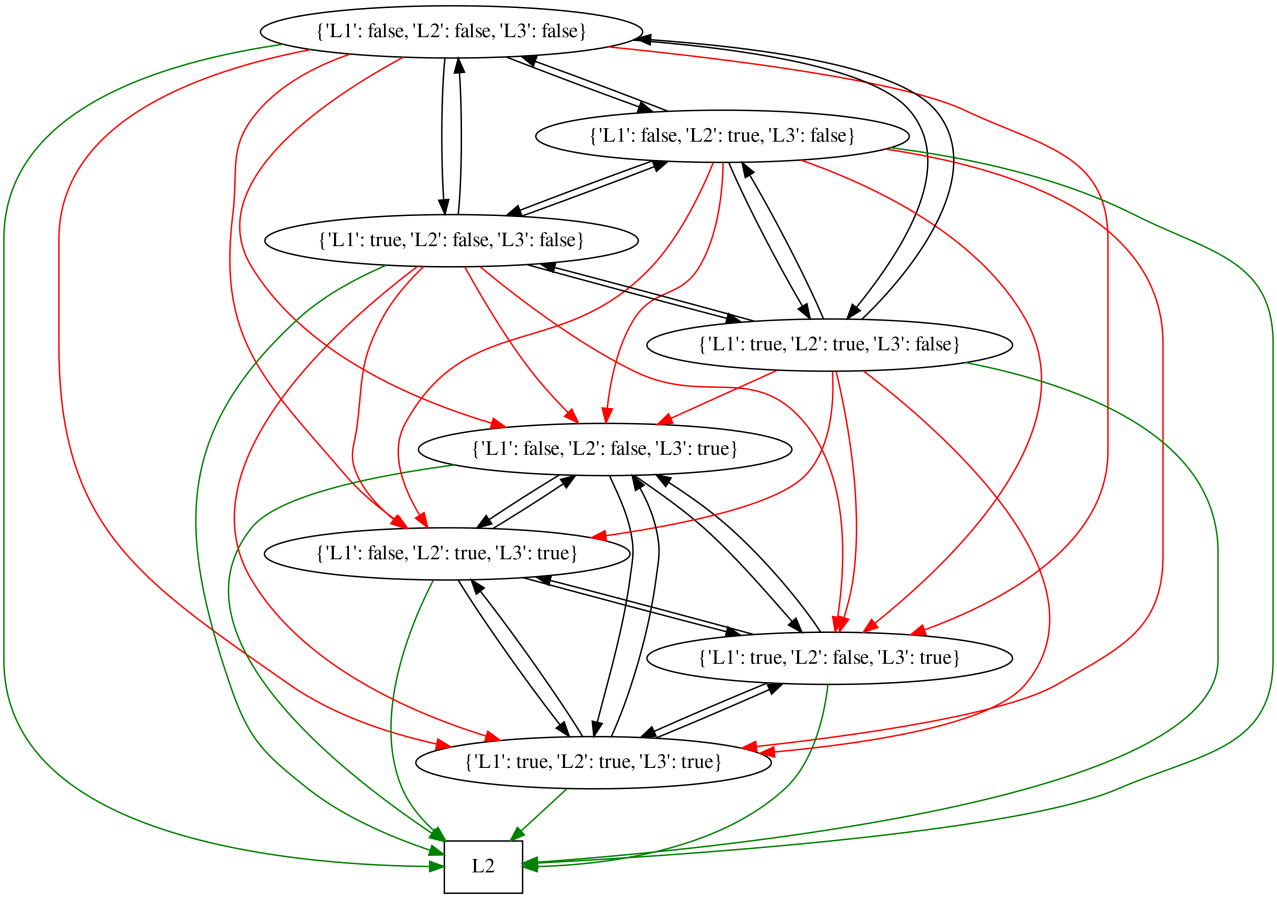


Figure 9. Accessible state graph from the light example obtained with the databases.

4.3. Discussion

This method allows efficient storage of the combinatorial explosion storing induced by the nature of the PLC. It also allows us to find information in tens of Gigabits of data. Once the generation of the marking graph is accessible, the retrieval of information from the database is rather fast. Its execution time is very long, and the code termination is not certain. This is a problem; in case of a simple error in the code, one can wait for the results for several days to find out that there is a mistake which requires a complete new exploration of the graph of markings. So, one solution could be to propose a method for a quick verification method that is less precise but that could indicate obvious errors.

5. Code Analysis by a Recursive Neural Network

5.1. Principle

We propose in this section an exploratory approach around the use of recurrent neural networks for the automatic verification of PLC code, the idea being not to have to browse the graph of reachable markings but only the code. The difficulty of implementing this method lies in the generation of a large database of checked model codes, which must be representative of those to which we wish to apply the neural network. We will limit our experiments to codes with few variables represented by a list of Boolean equations, so we can already validate the use of neural networks in a case where the generation of the database is not restrictive.

In order to process the codes with neural networks, we first need to convert them into a sequence of number vectors. We chose to put the codes in an elementary form by decomposing all the Boolean equations into a sequence of simple operations. The lines readable by the neural network must be compatible with the following regular expression

where the following are noted: $\backslash w+$ the variable names | the operator or and ? the presence of zero or one occurrence of the expression preceded by the operator.

$$(\backslash w+) \;:=\; (((\backslash w+)\ (\text{AND}|\text{OR})\ (\backslash w+))|((\text{NOT }){?}(\backslash w+)))\ ; \tag{1}$$

An example of code corresponding to this description is used in Figure 10. This format allows us to convert each line of the elementary code into a vector of $\{0,1\}^{2\times n_s+n_e+4}$ with n_e and n_s, respectively, the numbers of input and output variables of the codes. Line to vector conversions are given in Table 3.

We consider the internal variables as output variables because they have the same role in the construction of the graph of reachable markings. The n_s first coordinates of the vector are used to encode the output variable to which a new value will be assigned; they are all zero except for the coordinate corresponding to the variable number. The $n_s + 1 + n_e$ next corresponds to the variables appearing in the assignment and the next three are for the logical operator in the line.

Table 3. Examples of ST code lines and their encoding.

	S1	**S2**	**...**	**Sn_s**	**S1**	**S2**	**...**	**Sn_s**	**Init**	**E1**	**...**	**En_e**	**AND**	**OR**	**NOT**
S2 := E1 OR S1;	0	1	...	0	1	0	...	0	0	1	...	0	0	1	0
S1 := NOT Init;	1	0	...	0	0	0	...	0	1	0	...	0	0	0	1
S2 := Sn_s;	0	1	...	0	0	0	...	1	0	0	...	0	0	0	0
Sn_s := En_e AND En_e;	0	0	...	1	0	0	...	0	0	0	...	1	1	0	0

Now that we have a method to transform our ST codes into a sequence of number vectors, we can build binary classifiers based on a sequential path of the code using recurrent neural networks, and more particularly LSTM networks [25], which are widely used for sequential data analysis. Examples of their uses for source code processing problems can be found in the scientific literature, for example in [23,27,28] for the detection of different types of programming errors. We will jointly train an LSTM network and a multilayer perceptron to classify ST codes. The vectors x_k corresponding to each line of code will be given sequentially as input to the LSTM cell to update the memory vectors (h_k, c_k), and then the last short-term memory will be decoded by the multilayer perceptron to create the prediction vector whose each component between zero and one can be interpreted as a probability that the corresponding variable is involved in a fugitivity or deadlock problem in the ST code (Figure 11). During training, the parameters of the LSTM network and the perceptron will be optimized to minimize the cross-entropy between the objective and the prediction, which is the error function commonly used for binary classification problems.We could also train another LSTM network to classify the codes but this time by browsing them in the opposite direction. This would allow us to give more importance to the first lines of the codes, which would be the last ones to be analyzed by the LSTM network and thus would have the most influence on the output memory at the beginning of the training. This is the idea behind the classical bi-LSTM architectures (for bidirectional LSTM networks), where a forward LSTM and a backward LSTM browse the code independently and for which the multilayer perceptron (MLP) performing the classification takes as input the concatenation of the final memories of these two LSTM.

The LSTM networks browse the code sequentially; they are given the input lines in order of their appearance in the code. This way of browsing the code is not optimal because in a code there can be independent blocks of instructions whose order can be changed without affecting the output of the algorithm. The idea of our approach is to inscribe this invariance by permutation directly in the structure of the recurrent network . The idea of constructing LSTM networks that traverse a text in a non-sequential way is not new: in [26], LSTM networks based on the traversal of a tree rather than a sequence are presented.

However in our case, the structure adapted to code traversal is not that of a tree but that of a directed graph.

We can summarize the dependencies of the lines on the others in a directed graph that we will call the causality graph. Each node of this graph corresponds to a line of code, and there is an edge from one node to another if and only if the line corresponding to the arrival node involves a variable whose last assignment took place on the line corresponding to the departure node. We add a start node and an end node to this graph: the start node will be linked to the nodes having no other antecedents in the graph (which correspond to the lines of code that can be executed first because they do not depend on any other), and in an analogous way, the end node will have as antecedents all the nodes having no other successors (i.e., the nodes corresponding to the lines that can be executed last). To understand this, we use an example of arbitrary code and the associated causal graph (Figure 10). The start and end nodes are, respectively, numbered by 0 and -1. Line 9 uses the variable $s0$ which is modified for the last time in Line 2. It could therefore be executed at any position in the code as long as it is executed after Line 2 and before Line 10 since the latter depends directly on it. Similarly, Line 3 must be executed after Line 2 which itself must be executed after Line 1, but Lines 4, 5, 9 and 10 are not in the same branch as Line 3 and can be executed either before or after it. There is thus a multitude of codes associated with the same causal graph (except for the numbering of the nodes).

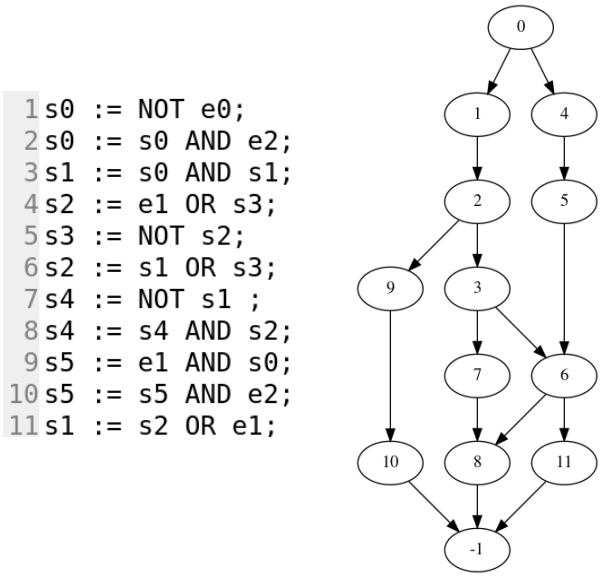


Figure 10. Example ST code and associated causal graph.

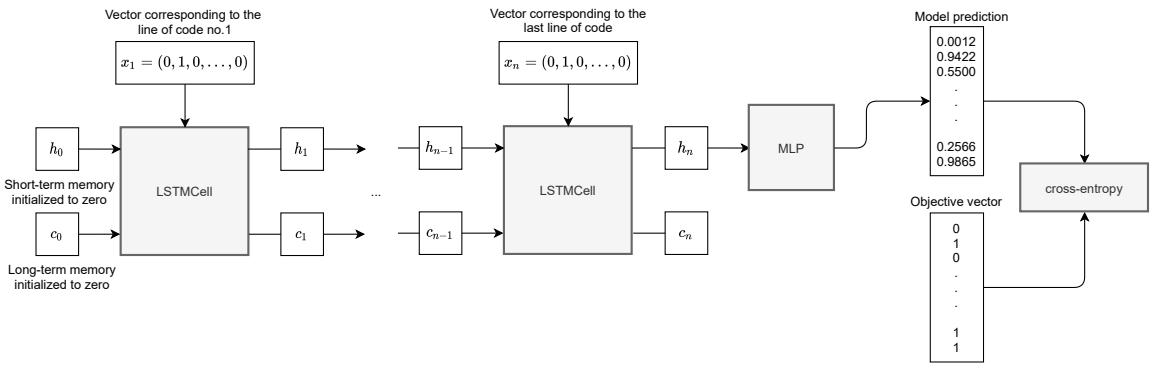


Figure 11. Simplified scheme for classifiers based on an LSTM network and a multilayer perceptron.

We define a DGLSTM (DiGraph long short-term memory) network as a network composed of an LSTM cell that is recurrently evaluated by following a given oriented graph which must verify some properties:

- It must have $n + 2$ nodes if the input sequences are of size n;
- It must have a single node without antecedents as well as a single node without successors;
- It must not contain any oriented cycle.

When a node has several antecedents in the graph, the memory used by the LSTM cell when evaluating the corresponding input will be the average of the memories computed in the output of the LSTM cell when evaluating the inputs corresponding to the antecedents of the node. The multilayer perceptron trained jointly with the DGLSTM network to make the prediction will take the average of the input memories of the output node as input . At each evaluation of a DGLSTM network, we must therefore not only provide the input sequence $(x_1, x_2, ..., x_n)$ but also the graph of causality of the corresponding code (Figure 12). In a similar way to bi-LSTM, we can define the bi-DGLSTM architectures for which the causality graph is browsed in both directions independently by two DGLSTM networks.

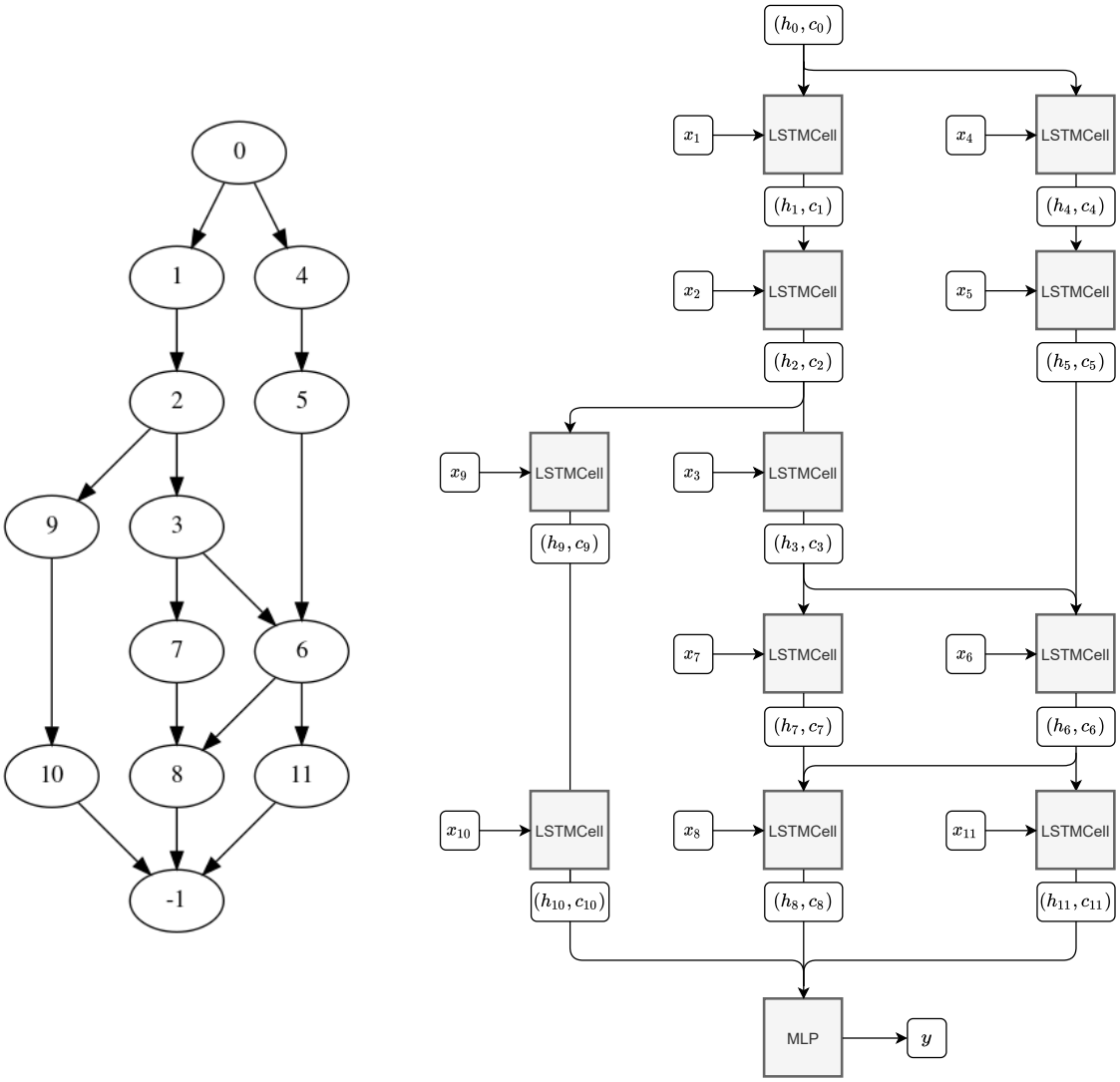


Figure 12. Causality graphs and associated DGLSTM network.

5.2. Experiments

We first generated a large number of random codes for which we set the maximum amount of input and output variables to 3 and 10, respectively. The lines of ST codes

are generated randomly and independently; there are between 5 and 15 per code. We used 250,000 codes for the training and 50,000 codes for the verification of the epoch. We used 50,000 other codes that had never been presented to the different neural networks to perform our statistical analysis. The fact that the lines are generated independently of each other means that there are many unnecessary lines of code. For example, it is common to assign a new value to a variable without having used it since its last assignment, but there are also many less trivial cases of unnecessary lines that can hardly be corrected. We do not really know to what extent the code used in practice by the APIs may contain unnecessary lines or unnecessarily complex portions of the code. In any case, we have chosen to keep the generation of codes by independent lines for the moment, hoping that the "realistic" codes are not under-represented in the set of codes that can be generated. The ideal solution would be to simplify the codes before they are processed by the LSTM or DGLSTM networks, but this would require considerable work. This theme of simplification of the codes has been evoked, especially in [20].

We trained five different types of models:

- LSTM networks based on a word encoding;
- LSTM networks based on line encoding;
- Bi-LSTM networks based on a line encoding;
- DGLSTM networks based on a line encoding;
- Bi-DGLSTM networks based on a line encoding.

All models were trained on the same amount of data and with batches of 128 codes. We used the ADAM algorithm [31] for training. RMSprop and SGD were tested without much success. We also used a strategy to reduce the training rate when the error decrease stopped and stopped training when the average error on the validation data stopped decreasing due to over-learning. We used a Bayesian optimization algorithm of the library *optuna* to find the best set of hyperparameters for each of the five types of models. The error on the validation data at the end of training was the criterion to minimize. The hyperparameters in question are the same for all five types of models:

- Size of the memory of the LSTM cells;
- Number of LSTM cells composed during the evaluation of a single input vector;
- Number of layers and number of neurons of the multilayer perceptron;
- Value of the dropout layers.

In addition to the cross-entropy, four other error measures are defined which are only used to analyze the results: the percentages of false positives and false negatives for the deadlock and for the fugitivity. The error on the validation base decreases faster for DGLSTM and bi-DGLSTM than for the other models: a few epochs are usually enough to reach a plateau in the error, while it takes several tens for the others. This plateau generally corresponds to a lower error than for the other models. For both (bi-)LSTMs and (bi-)DGLSTMs, the deepest networks give the best results but at the cost of a more difficult training which seems very sensitive to the initialization weights. Sometimes, the error does not decrease at all even for a large number of epochs, whereas it sometimes decreases from the first epochs for identical hyperparameters. Thus, the best results we have obtained are the result of good luck at initialization.

In addition to the cross-entropy, four other error measures are defined which are only used to analyze the results: the percentages of false positives and false negatives for the deadlock and for the fugitivity (Figure 13). It seems that it is more difficult to predict deadlock than fugitivity. We ran the hyperparameter optimization routine for the same amount of time for all five model types. The models with the slowest evaluation could therefore be trained a smaller number of times than the others. The (bi-)DGLSTM networks take by far the longest to evaluate and are therefore the most disadvantaged for this choice, which does not prevent them from having the best results, thus demonstrating the relevance of the non-sequential browsing of the ST codes.

We have demonstrated the ability of neural networks to make correct predictions for the majority of randomly generated codes, but this does not guarantee that these predictions will always be mostly correct for real codes used in practice. We therefore applied our best bi-DGLSTM networks to model check the three-lamp example. We have therefore evaluated the best DGLSTM network on the three-lamp example (Table 4). Note that the criterion for selecting the best DGLSTM is the prediction error on the validation base composed of random codes. The network was therefore not selected to be the best on this particular example, which would have led to artificially good results. The network succeeds almost perfectly in predicting fugitivity, with only one minor error for the second lamp. As observed, on average on all random codes, the prediction of deadlock seems much worse.

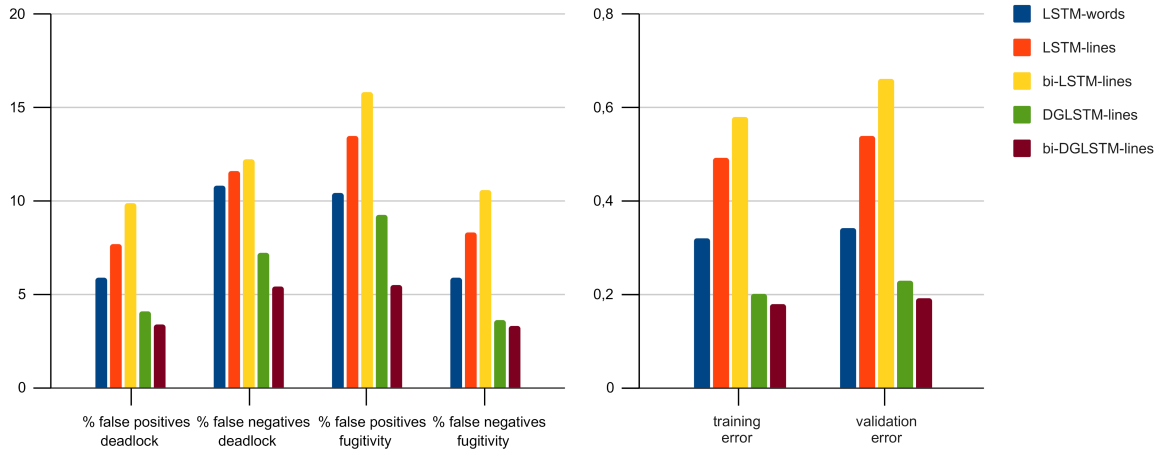


Figure 13. Error obtained for the best model of each type.

Table 4. Results obtained with the best DGLSTM on the validation set for the 3-lamp example.

	Predicted Deadlock	**Expected Deadlock**	**Predicted Fugitivity**	**Real Fugitivity**
FTX10	0.057	0	0.004	0
FTX11	0.081	0	0.899	1
X10	0.931	0	0.015	0
X11	0.571	0	0.211	0
FTX20	0.308	1	0.387	0
FTX21	0.011	0	0.972	1
X20	0.718	0	0.079	0
X21	0.166	1	0.667	0
FTX30	0.043	0	0.937	1
X30	0.992	0	0.002	0
X31	0.088	0	0.842	1

5.3. Discussion

The use of neural networks for model checking seems promising. A full-fledged study would be needed to improve the performance of the networks and extend their use to codes with larger numbers of variables, which would require more attention to parsimonious database generation. In particular, we did not conduct any data augmentation. From a single code verified with UPPAAL, it would have been possible to create many equivalent codes that do not necessarily have the same causal graphs, for example by permuting the names of some variables. Finally, it would be interesting to understand the codes that cause problems to neural networks. The use of the causal graph to browse the code in a non-sequential way seems relevant and is not restricted to the context of our study.

Table 5 presents, as a synthesis, the results obtained from the three methods proposed in the paper. From these first results, we could imagine for the future the following testing PLC code methodology. Once a compilable code is obtained, it is best to start with an algorithm based on syntactic analysis by neural networks that provides a quick opinion on the provided code. This operation allows detecting faults as soon as possible, which allows the engineer to correct them before using more resource-intensive methods. Then, we can use UPPAAL or databases depending on the size of the code to certify that the errors defined in this article are in the code. The key point of our proposed method is to find simple errors in a code that are often programming errors without a priori. Thus, all the codes implemented in a factory should go through these verification steps to eliminate simple faults which would considerably reduce the time of analysis with respect to the behavior of the PLC which requires a comparison with the driven machine and its environment.

Table 5. Comparison of the different methods presented in this paper.

	UPPAAL-Based Verification	**Database-Based Verification**	**Neural Network Verification**
Understood semantics	Full ST code	90% of ST code	Boolean operation
Maximum number of variables	Around 20	Around 200	8
Execution time	20 min for 20 variables codes	12 h for 150 variables codes	0.5 s
Validity of results	Certain if completion	Certain if completion	Uncertain

6. Conclusions

In this article, we have discussed several methods that could allow the Factory 4.0 engineers to test generic properties of PLC programs in a simple way. To do this, we explored different methods that could be used together in the future for testing PLC programs of different sizes and complexities. Moreover, we believe that this work could be used as a basis for more complex database implementations on supercomputers and for training neural networks on larger syntax. Indeed, the exploration of the accessible marking graph can be easily parallelized, and the accessibility to the database can be parallelized. A work on an implementation closer to the functioning of a database would certainly make it possible to treat much more complex cases in record time. Furthermore, neural networks are able to translate and summarize text thanks to attention mechanisms which could certainly give excellent results on code verification. At the moment, our algorithm is not able to cover all the codes implemented on the different controllers used in the industry. Some complex semantics inducing strong combinatorial explosion cannot be present in the verified codes. We hope to be able to extend our work to the whole set of codes by implementing other ingenious solutions allowing us to treat the various combinatorial problems.

The final goal of this study is to support humans to design reliable PLC codes. During this first study, we did not submit our methods to actors who did not know our research subject. One of the objectives of our future work will be to submit our proposed tools to personnel working on the realization of PLC code in an industrial environment.

Author Contributions: Conceptualization, methodology, software, validation, writing—original draft preparation, review and editing É.S., L.P.; investigation and formal analysis T.R.; supervision, project administration, investigation, writing—review and editing B.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Acknowledgments: First, we would like to thank Jean-Marc Roussel who helped us to understand the formal verification method on PLC and David Annebicque who helped us to efficiently implement graphs on databases. Secondly, we would like to thank Vincent Lepetit for his point of view on

neural networks. Finally, we thank the team of the French HPC Center ROMEO which allowed us to use its cluster, thus obtaining interesting results.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DGLSTM	DiGraph Long Short-Term Memory
FAT	Factory Acceptance Testing
FPGA	Field-Programmable Gate Array
Grafcet	Graphe Fonctionnel de Commande des Étapes et Transitions (Sequential Function Chart)
IT	Information Technology
LSTM	Long Short-Term Memory
MLP	MultiLayer Perceptron
OT	Operational Technology
PLC	Programmable Logic Controllers
RAM	Random Access Memory
SAT	Acceptance Testing
SQL	Structured Query Language
ST	Structured Text

References

1. Vogel-Heuser, B.; Diedrich, C.; Fay, A.; Jeschke, S.; Kowalewski, S.; Wollschlaeger, M.; Göhner, P. Challenges for Software Engineering in Automation. *J. Softw. Eng. Appl.* **2014**, *7*, 440–451. [CrossRef]
2. Sehr, M.A.; Lohstroh, M.; Weber, M.; Ugalde, I.; Witte, M.; Neidig, J.; Hoeme, S.; Niknami, M.; Lee, E.A. Programmable Logic Controllers in the Context of Industry 4.0. *IEEE Trans. Ind. Inform.* **2021**, *17*, 3523–3533. [CrossRef]
3. Hajda, J.; Jakuszewski, R.; Ogonowski, S. Security Challenges in Industry 4.0 PLC Systems. *Appl. Sci.* **2021**, *11*, 9785. [CrossRef]
4. Langmann, R.; Stiller, M. The PLC as a Smart Service in Industry 4.0 Production Systems. *Appl. Sci.* **2019**, *9*, 3815. [CrossRef]
5. Calderón Godoy, A.J.; González Pérez, I. Integration of Sensor and Actuator Networks and the SCADA System to Promote the Migration of the Legacy Flexible Manufacturing System towards the Industry 4.0 Concept. *J. Sens. Actuator Netw.* **2018**, *7*, 23. [CrossRef]
6. Niang, M.; Riera, B.; Philippot, A.; Zaytoon, J.; Gellot, F.; Coupat, R. A methodology for automatic generation, formal verification and implementation of safe PLC programs for power supply equipment of the electric lines of railway control systems. *Comput. Ind.* **2020**, *123*, 103328. [CrossRef]
7. Pichard, R.; Philippot, A.; Saddem, R.; Riera, B. Safety of Manufacturing Systems Controllers by Logical Constraints With Safety Filter. *IEEE Trans. Control Syst. Technol.* **2019**, *27*, 1659–1667. [CrossRef]
8. Zaytoon, J.; Riera, B. Synthesis and implementation of logic controllers—A review. *Annu. Rev. Control* **2017**, *43*, 152–168. [CrossRef]
9. Larsen, K.G.; Pettersson, P.; Yi, W. UPPAAL in a Nutshell. *Int. J. Softw. Tools Technol. Transf.* **1997**, *1*, 134–152. [CrossRef]
10. Cimatti, A.; Clarke, E.; Giunchiglia, F.; Roveri, M. NUSMV: A new symbolic model checker. *Int. J. Softw. Tools Technol. Transf.* **2000**, *2*, 410–425. [CrossRef]
11. Malik, R.; Åkesson, K.; Flordal, H.; Fabian, M. Supremica–An Efficient Tool for Large-Scale Discrete Event Systems. *IFAC-PapersOnLine* **2017**, *50*, 5794–5799. [CrossRef]
12. Machado, J.; Denis, B.; Lesage, J.J. Formal Verification of Industrial Controllers: With or without a Plant model? In Proceedings of the 7th Portuguese Conference on Automatic Control, Lisbon, Portugal, 11–13 September 2006 .
13. Mendes, M.J.; Eurico, S.; Creissac, C.J.; Filomena, S.; Pinto, L.C. Simulation and formal verification of industrial systems controllers. *ABCM Symp. Ser. Mechatron.* **2008**, *3*, 461–470.
14. Ovsiannikova, P.; Buzhinsky, I.; Pakonen, A.; Vyatkin, V. Oeritte: User-Friendly Counterexample Explanation for Model Checking. *IEEE Access* **2021**, *9*, 61383–61397. [CrossRef]
15. Bonfe, M.; Fantuzzi, C. Design and verification of mechatronic object-oriented models for industrial control systems. In Proceedings of the EFTA 2003 IEEE Conference on Emerging Technologies and Factory Automatio, Lisbon, Portugal, 16–19 September 2003; Volume 2; pp. 253–260. [CrossRef]
16. Tang, L.; Ma, G. A Quick Modeling Approach in Model Checking. In Proceedings of the 2013 Seventh International Conference on Internet Computing for Engineering and Science, Shanghai, China, 20–22 September 2013; pp. 89–94. [CrossRef]

17. Xiong, J.; Zhu, G.; Huang, Y.; Shi, J. A User-Friendly Verification Approach for IEC 61131-3 PLC Programs. *Electronics* **2020**, *9*, 572. [CrossRef]
18. Provost, J.; Roussel, J.M.; Faure, J.M. A formal semantics for Grafcet specifications. In Proceedings of the 2011 IEEE International Conference on Automation Science and Engineering, Trieste, Italy, 24–27 August 2011; pp. 488–494. [CrossRef]
19. Provost, J.; Roussel, J.M.; Faure, J.M. Translating Grafcet specifications into Mealy machines for conformance test purposes. *Control Eng. Pract.* **2011**, *19*, 947–957. .: 10.1016/j.conengprac.2010.10.001. [CrossRef]
20. Guignard, A.; Faure, J.M.; Roussel, J.M. Génération d'une machine de Mealy à partir de spécifications algébriques à des fins de test de conformité. In Proceedings of the Conférence Internationale Francophone d'Automatique (CIFA2012), Lyon, France, 4–6 July 2012; pp. 907–912.
21. Guignard, A.; Faure, J.M.; Faraut, G. Model-Based Testing of PLC Programs With Appropriate Conformance Relations. *IEEE Trans. Ind. Inform.* **2018**, *14*, 350–359. [CrossRef]
22. Qanadilo, M.; Samara, S.; Zhao, Y. Accelerating Online Model Checking. In Proceedings of the 2013 Sixth Latin-American Symposium on Dependable Computing, Rio de Janeiro, Brazil, 1–5 April 2013; pp. 40–47. [CrossRef]
23. Dam, H.K.; Tran, T.; Pham, T. A deep language model for software code. *arXiv* **2016**, arXiv:1608.02715.
24. Minaee, S.; Kalchbrenner, N.; Cambria, E.; Nikzad, N.; Chenaghlu, M.; Gao, J. Deep Learning Based Text Classification: A Comprehensive Review. *arXiv* **2021**, arXiv:2004.03705.
25. Olah, C. Understanding LSTM Networks. 2015. Available online: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (accessed on 1 January 2023) .
26. Tai, K.S.; Socher, R.; Manning, C.D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *arXiv* **2021**, arXiv:1503.00075.
27. Rahman, M.; Watanobe, Y.; Nakamura, K. Source Code Assessment and Classification Based on Estimated Error Probability Using Attentive LSTM Language Model and Its Application in Programming Education. *Appl. Sci.* **2020**, *10*, 2973. [CrossRef]
28. Hajiaghayi, M.; Vahedi, E. Code Failure Prediction and Pattern Extractionusing LSTM Networks. *arXiv* **2018**. arXiv:1812.05237.
29. Fernández Adiego, B.; Darvas, D.; Viñuela, E.B.; Tournier, J.C.; Bliudze, S.; Blech, J.O.; González Suárez, V.M. Applying Model Checking to Industrial-Sized PLC Programs. *IEEE Trans. Ind. Inform.* **2015**, *11*, 1400–1410. 2489184. [CrossRef]
30. Roussel, J.M.; Lesage, J.J. Design of Logic Controllers Thanks to Symbolic Computation ofSimultaneously Asserted Boolean Equations. *Math. Probl. Eng.* **2014**, *2014*, 726246. [CrossRef]
31. Kingma, D.P.; Ba, L. ADAM: A Method for Stochastic Optimization. *arXiv* **2017**, arXiv:1412.6980.