Towards automatic verification of ladder logic programs

Bohumír Zoubek
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT
United Kingdom
Email: bxz@cs.bham.ac.uk

Jean-Marc Roussel
LURPA
Ecole Normale Superieure De Cachan
61, Avenue du Président Wilson
94235, Cachan Cedex, France
Email: roussel@lurpa.ens-cachan.fr

Marta Kwiatkowska
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT
United Kingdom
Email: mzk@cs.bham.ac.uk

***Abstract*— Control system programs are usually validated by testing prior to their deployment. Unfortunately, testing is not exhaustive and therefore it is possible that a program which passed all the required tests still contains errors. In this paper we apply techniques of automatic verification to a control program written in ladder logic. A model is constructed mechanically from the ladder logic program and subjected to automatic verification against requirements that include timing. This consists of an exhaustive search of the model of the program, thus eliminating the drawback of testing. We believe that automatic verification can substantially enhance current validation procedures for control programs.**

I. Introduction

Control systems are used in many applications of process control all around the world. Their uses range from small applications such as building alarm systems to larger applications such as production factories and nuclear power plants. Any fault can cause injury or loss of life and environmental or economical damage. Factory downtime in all cases only adds to the overall cost of the failure.

Programs for control systems are currently validated by testing. A set of tests for each program is executed in order to establish the compliance or otherwise of the program with its specification. The main disadvantage of such program testing is that it is not exhaustive, which creates the possibility of a program being successfully tested and still containing errors. This risk can be reduced by verification of control programs. This paper demonstrates the application of automatic verification techniques on a small control program written in ladder logic against non-trivial specifications.

The framework introduced in this paper is intended as an add-on extension to a typical control programming toolbox. We focus on ladder logic, one of the standardized programming languages for Programmable Logic Controllers. After a ladder logic program has been developed with the help of the toolbox in the usual way, it is input by our software tool which first performs a transformation of the program into a model known as a timed automaton. The model is then entered into a pre-existing tool called model checker. The model checker accepts specifications and explores the model in order to establish if each specification holds; if not, a diagnostic trace is produced. The central idea is to improve reliability of control programs by offering the programmer the functionality of checking for desirable conditions, using notations readily understood in the control engineering domain, and hiding the complexity of model construction inside the tool itself.

II. Terminology and formalisms

In this section we introduce the domain of application for our methodology, and explain the main concepts and formalisms used.

A. Programmable Logic Controller

Programmable Logic Controllers (hereafter PLCs) are used as a hardware platform on which a control program is executed, and are connected to the process unit via an input/output system. PLCs are, in fact, industrial computers specifically designed to be used as control systems.

Programs for PLCs can be written in one or more languages standardized by IEC 61131-3 [1] and are executed in cycles with three main steps (Fig. 1). First, the inputs of the control system are read and their values stored in memory. Then the program is executed using the stored input values, and all computed values of outputs are also stored. As the last step, all outputs are activated based on their values in the memory. The length of a PLC cycle is called the PLC scan.

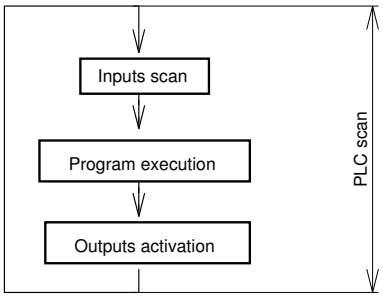

Fig. 1
PLC program execution cycle

B. Ladder logic

Ladder logic, one of the standardized languages for PLCs, is a graphical programming language with a representation similar to that of circuit diagrams. Programs written in ladder

logic look like ladders, with commands written into rungs, and are executed rung by rung. Commands are represented by means of symbols of the electrical engineering domain and the complete set of instructions varies slightly from one manufacturer to another. An example of a rung can be seen in Fig. 2.

One way to think about ladder logic is that the left side of the ladder is subjected to a power source, and this power is trying to reach the right side of it along each rung. The pictured rung consists of two instructions (left to right): *examine if closed* and *output energize*. The first instruction checks the value of the `L1_PROD` variable, and if true it allows the power through. The second instruction sets the value of `L1_UP` to true, if it can be reached by the power, or to false otherwise. For our purposes we define two groups of ladder logic instructions: *conditions* and *actions*. The first instruction in the above is a condition and the second one is an action.

Fig. 2
A LADDER LOGIC RUNG

Rung continuity arises if the power source is able to reach the right-hand side of the rung. Such a rung is said to be *continuous*.

C. Timed automata

The timed automata formalism [2] is a modelling language that allows us to describe the behaviour of systems by means of finite state automata diagrams extended with clocks and time constraints. For example, consider a model of a system which consists of three states *A, B, C* and whose possible transitions are $A \longrightarrow B$, $B \longrightarrow C$, $C \longrightarrow A$ and $C \longrightarrow B$. We obtain the finite state automaton shown in Fig. 3.

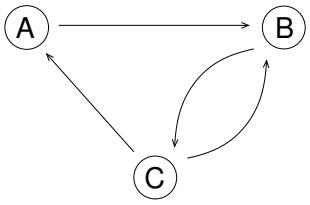

Fig. 3
FINITE STATE AUTOMATON

Such a system can be extended with clocks and time constraints. A *clock* of the system is a variable that is increased at the same rate as time. Clocks can be reset or assigned some new (integer) value. Time constraints are based on predefined clocks and can be of two types. *Invariants* define a time interval for which the system can remain in a state and *guards* enable transitions within a time interval.

To illustrate the main concepts, we impose time constraints based on clock c on the finite state automaton of Fig. 3. The automaton is to behave periodically in the interval $c \in [0,\ 15)$ in such a way that it may stay in *A* when $c \in [0,\ 10)$, it may move to *B* if $c \geq 5$, it may stay in *B* if $c < 15$ and it may stay in *C* as long as $c < 5$. Moreover, the clock c is reset when the system takes the transition $B \longrightarrow C$.

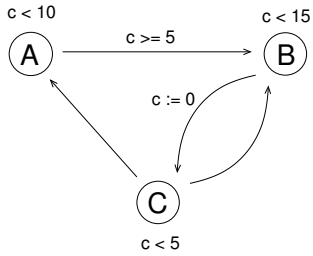

Fig. 4
TIMED AUTOMATON

Such a timed automaton is shown in Fig. 4. Note that invariants are constraints over states, whereas guards are constraints over transitions. There is one assignment used to reset the clock c during the transition $B \longrightarrow C$. Transitions without guards can be taken at any time.

D. Model checker

A model checker is a software tool that can be used to *automatically* establish whether the model satisfies its specification or not. In the latter case it will also produce a trace showing where the specification is breached. We use the *real-time model checker* Uppaal [3]. Uppaal accepts a timed automaton model and the model specification as inputs, checks the compliance of the model to the specification and outputs the results including diagnostic information. The specifications are given in Uppaal specification language with the help of two prefixes: `A[]` and `E<>`. A property prefixed by the former is satisfied if it holds on all paths of the model; the latter prefix indicates that the model can reach a state where the property holds. Some examples of the property specifications in the Uppaal language are:

- `A[] (t2-t1)`$\leq$`100`
  Condition `(t2-t1)`$\leq$`100` holds in all states, on all paths of the system (invariantly). In this particular case, `t1` and `t2` are clocks, and therefore the condition expresses the time difference no greater than 100.
- `E<> CH1.synchro`
  The system can reach the state `CH1.synchro` (via some path, after a finite number of steps).

The Uppaal models are slight variants of standard timed automata, and in particular allow certain convenient modelling features which we now outline. Three additional types of states (fig. 5) are available for use. The *initial* state defines the position of an automaton at the beginning (when the time is equal to zero). A *committed* state is one which has to be left immediately. *Urgent* states are used to prevent a delay when a transition is already enabled and when more than one transition (to more than one state) is enabled at the same time.

E. The Checker tool

We have implemented the algorithms presented in this paper in the Checker tool which is written in Java. It takes

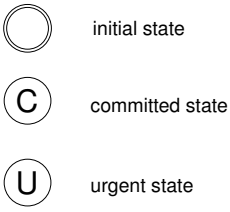

Fig. 5
ADDITIONAL TYPES OF STATES IN UPPAAL TIMED AUTOMATA

a ladder logic control program as its input and mechanically builds a timed automaton model for it. Such a model is then verified using Uppaal, possibly after further transformations of the model have been performed for efficiency reasons. We have employed the Checker tool on a pumping line unit case study previously verified by manual theorem proving, using a calculus developed for this purpose, in [4]; in contrast, our method is fully automatic.

III. Related work

This section briefly outlines the relevant work done by other people in the field of control program validation. The use of formal methods in PLC programming is introduced in [5] and there are two other modelling languages, besides timed automata [6], that have been used to model control programs: PLC-automata [7], [8] and Condition/Event Systems [9], [10].

Moby/PLC [11] is a graphical tool that can be used to build PLC-automata models of control programs. The tool also supports simulation and export of the model into formats accepted by Uppaal or Kronos [12]. Similarly, Verdict [13] is a graphical tool for Condition/Event Systems. Both tools require the knowledge of the modelling language, in contrast to our approach where this knowledge is not needed.

The VHS project [14], a collaboration among several European partners, concentrated mainly on the design correctness [15] of control systems such as a manufacturing plant. In [16] the Uppaal model checker was applied in order to verify the batch plant.

IV. Case study

We use a control program for a pumping line unit which provides a water supply from a tank via two pumps (shown in Fig. 6), originally introduced in [4], as our case study.

The unit consists of the water tank, two pumps, the backflow valve and the output valve. In addition to these components there are further two valves, the upstream valve and the downstream valve, associated with each pump. The unit is expected to operate in such a way that safety requirements are not infringed. Safety properties for our case study are shown in table I.

A control system interaction with the unit is provided by inputs and outputs. The list of inputs and outputs for our case study is in table II.

V. Our approach

Our work is motivated by the idea that a control system programmer should be able to automatically verify a program prior to its deployment. We achieve this by mechanically translating control programs into timed automata models and then by verifying these models using Uppaal. The main challenge in this work is finding efficient ways of translating programs into compact, manageable models. This is done through a number of transformations, shown in Fig. 7 and described below.

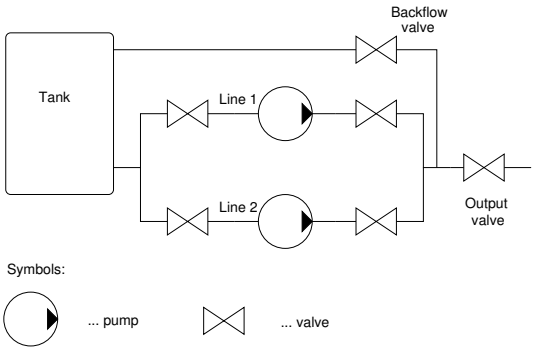

Fig. 6
PUMPING LINES SCHEMA

P1	The upstream valve of a pump must be opened for 5 seconds prior to the pump startup.
P2	The two pumping lines must not work together.
P3	The backflow and the output valves must be closed if both pumps are not running.
P4	In case of a pump failure all associated actuators will be turned off.

TABLE I
CONTROL PROGRAM PROPERTIES IN NATURAL LANGUAGE

`L_flow`	Water distribution at low flow rate request
`H_flow`	Water distribution at high flow rate request
`Line_swap`	Change the line priority request
`L`i`_fail`	Fault information from pump i
`SP_fail`	Distribution has been stopped
`L`i`_pump`	Start pump i
`L`i`_up`	Open the upstream valve of pump i
`L`i`_down`	Open the downstream valve of pump i
`Out_valve`	Open the output valve
`Bf_valve`	Open the backflow valve

TABLE II
LIST OF INPUTS AND OUTPUTS

A. Control program and property

We have implemented our algorithms for a subset of ladder logic instructions used by Allen-Bradley PLCs of the SLC5/03 series. We have one PLC of this series on loan from Rockwell Automation Ltd together with RSLogix 500 programming toolbox. Properties are defined in the Uppaal specification language.

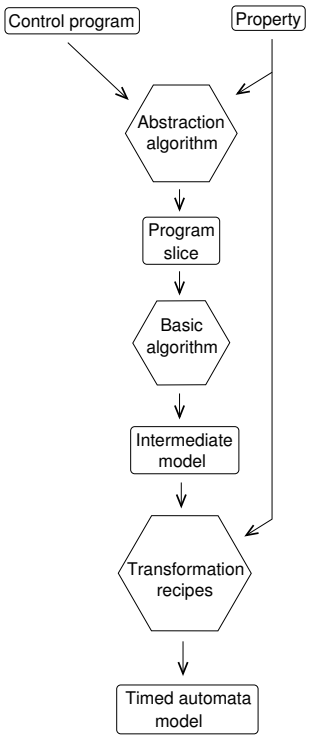

Fig. 7

BUILDING A TIMED AUTOMATON MODEL

B. Abstraction algorithm and program slice

A common approach to verification is to build a model of the complete control program first and subject it to verification of all properties required. This approach will result in a complex model and almost certainly the state space explosion problem would be encountered for many properties.

To deal with the state explosion problem we propose the use of a program slicing algorithm. We fix the property to be verified, and determine which part of the program affects it. This part, or "slice", of the program only is then transformed into the model. Such an approach works quite well for programs that contain many modules, each controlling a part of the system, as long as the property to be verified pertains to a subset of modules.

C. Basic algorithm and the intermediate model

The basic algorithm translates the program slice into a timed automaton model. It also generates other necessary components (timed automata) such as the PLC program execution and the input/output system. All these timed automata communicate with each other via synchronization channels, and their parallel composition yields the desired model of behaviour. We now describe the process in more detail.

1) Program slice: The program slice is translated rung by rung into a timed automaton model. This is where the notion of rung continuity, and the grouping of instructions into conditions and actions, play an important role. We will only outline the translation briefly, since a full description is beyond the scope of this paper.

Consider a condition instruction $x = 1$ positioned at the beginning of a rung. It will be translated into a timed automaton using two transitions: one will be taken when the condition is satisfied (i.e. $x == 1$) and the other one when it is not (i.e. $x! = 1$). The rung execution will either lead to the next instruction along the rung (if the condition is satisfied), or the rung will be completely skipped. The model of this instruction is shown in Fig. 8. We say that the second transition is a *negative* transition. When translating a rung with more than one condition instruction a number of negative transitions will be created. The importance of this notion will emerge when modelling action instructions as well as in transformation recipes.

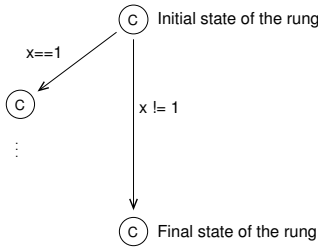

Fig. 8

MODEL OF A CONDITION INSTRUCTION

Let us now consider an action instruction placed at the end of the rung, which sets y to 1 if the rung is continuous and to 0 otherwise. It will be translated as follows: one transition from the previous state (i.e. the state when the previous transition ends) to the final state of the rung with an assignment $y := 1$. There will also be an assignment $y := 0$ on all previous negative transitions (these are created by modelling condition instructions such as the one described the last paragraph). The model of this instruction is shown in Fig. 9.

Intermediate instructions (those not at the beginning nor end of a rung) are dealt with similarly. Their corresponding models will be positioned in place of ... in figures mentioned above. The resulting Uppaal timed automaton of the rung from Fig. 2 is shown in Fig. 10. All states used in these models are of the committed type to ensure execution without delays.

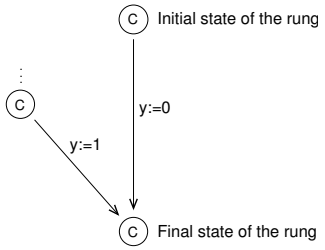

Fig. 9

MODEL OF AN ACTION INSTRUCTION

2) Inputs and outputs: Inputs have to be modelled as variables that change their values nondeterministically. This guarantees that every possible behaviour of the model will be checked during verification. Each input is usually modelled by a small timed automaton such as the one shown in Fig. 11.

This is a model of the input address `I1:0/3`. It uses a global variable `g_I_1_0_3` that can change its value nondeterministically at any point in time, but its value is copied into the local variable `I_1_0_3` of the program model during the input scan part of the PLC cycle. This local variable is then used during program execution.

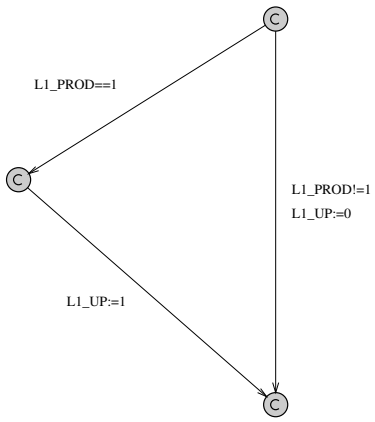

Fig. 10
MODEL OF THE RUNG FROM FIG. 2

There is no need for such constructs in the case of outputs. Outputs are modelled as variables used in the program model. However, all properties (and hence all values of outputs) are verified at the end of the program model execution in order to comply with the output activation part of the PLC scan.

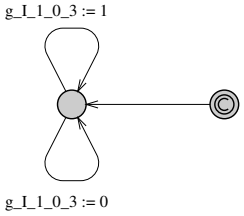

Fig. 11
MODEL OF THE INPUT

3) Program execution: The timed automaton in Fig. 12 is included in the intermediate model to achieve the cyclic program execution typical for PLCs (Fig. 1). Transition from state `waiting` to state `executed` requests synchronization via channel x which triggers the program execution in such a way that the whole model of the program is executed at the same time as this transition. Hence, the program is executed periodically in intervals given by the constant `scan`. This constant defines the length of a PLC scan, which was set to 20 msec. throughout these experiments.

D. Transformation recipes and the timed automata model

The structure of each rung can be represented by a graph. The basic algorithm builds models of all instructions and connects them via transitions according to rung structures. The resulting model can be further simplified, eg. by removing redundant transitions, in order to reduce the model size. Transformation recipes are used to achieve this.

In some cases the intermediate model needs to be enhanced further to enable verification. For example, in order to verify a timed property a timer must be added. This is achieved by an appropriate transformation recipe. Another case would be a branching based on the value of one variable. Again, there is a transformation recipe that serves this purpose.

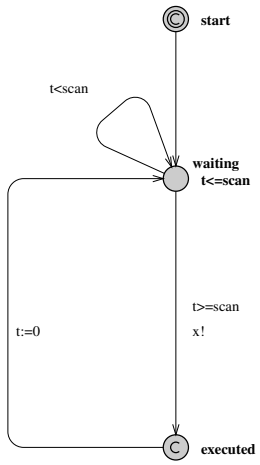

Fig. 12
MODEL OF THE PLC CYCLE

VI. VERIFICATION

This section describes the automatic verification task performed for the case study of section IV. We succeeded with all the properties shown in table I. In this section we present our findings and describe the technical difficulties encountered during in the process and ways to overcome them.

A. Program

The original program (Roussel and Denis [4]) makes use of bistable function blocks that are not supported by our PLC. Therefore these blocks have been translated into an equivalent combination of instructions supported by our PLC (Allen-Bradley). Bistable function blocks are of two types: set dominant and reset dominant (our case). All inputs and outputs of the block are digital variables (booleans). The blocks used in the original program and its body are shown in Fig. 13 and its ladder logic version from our program is shown in Fig. 14. The program size increases, as a result of the above translation, by 6 rungs. A complete listing of the control program that is subjected to verification in this paper is given in Figs. 16 and 17.

B. Property 1

This timed property (table I) states that a pump's own upstream valve must be open for 5 seconds prior to the pump being turned on. In general, timed properties are very difficult to verify because of the state explosion problem, and

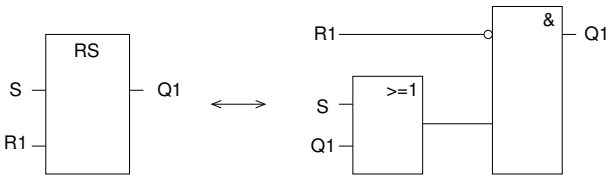

Fig. 13

BISTABLE FUNCTION BLOCK (RESET DOMINANT) AND ITS BODY

Fig. 14

BISTABLE FUNCTION BLOCK (RESET DOMINANT) IN LADDER LOGIC

indeed verification of this property on the model of the control program failed due to this. However, we were able to verify the property by the following means (explained here for pump 1, but the same technique works for both pumps because they are controlled by the corresponding parts of the program, although with different addresses).

1) Timed variable: First we select the timed variable which in the case of pump 1 is `L1_UP`. Property 1 describes the situation when the timed variable has value 1 (pump 1 upstream valve is opened by `L1_UP` being equal to 1) so we introduce an additional clock `c_L1_UP`, as a part of the timed automaton model of the program, that ticks while `L1_UP` $= 1$ and is reset when `L1_UP` $\neq 1$. This is achieved by resetting the clock on the first transition (program is searched from the top to the bottom) with guard `L1_UP` $= 0$ (or `L1_UP` $\neq 1$) or, if such a transition does not exist, on the first transition with assignment `L1_UP` $:= 0$. Then we redefine property 1 using this clock, i.e. `c_L1_UP` must be greater than or equal to 5 in order to start pump 1.

In our case only the transition with assignment `L1_UP` $:= 0$ can be found and the clock reset is added as shown in Fig. 15.

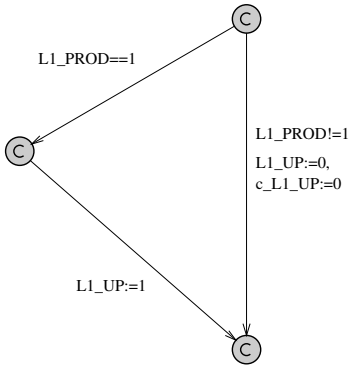

Fig. 15

MODEL OF THE RUNG FROM FIG. 2 WITH TIMED VARIABLE

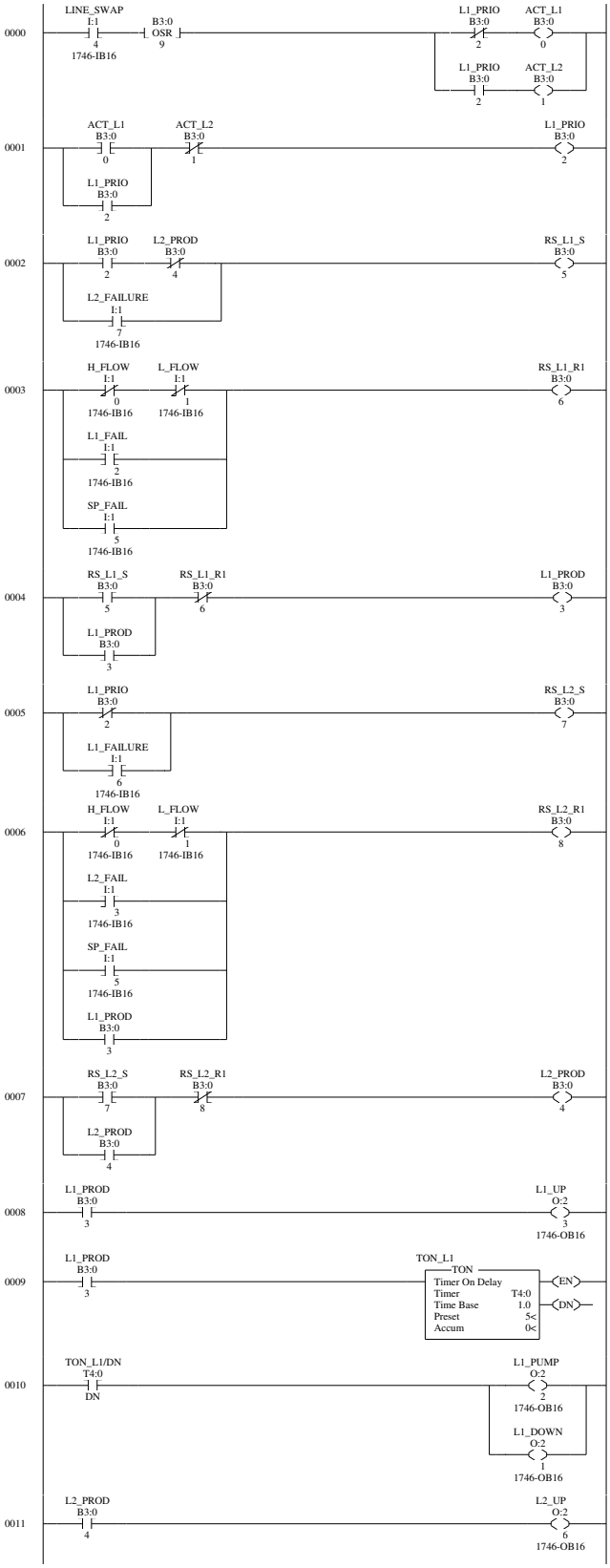

Fig. 16

CONTROL PROGRAM - PART 1

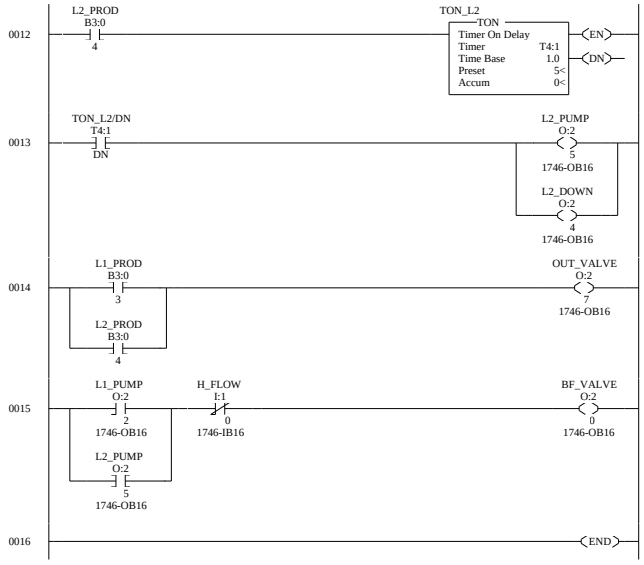

Fig. 17
CONTROL PROGRAM - PART 2

2) Abstraction: It can be seen by inspecting the program (Fig. 16) that this property is determined in rungs 8-10. `L1_UP`, which opens the upstream valve, is the output of rung 8 and set to 1 while variable `L1_PROD` is equal to 1. This variable is also timed in rung 9 and activates `L1_PUMP`, which starts pump 1, in rung 10. This means that a model of the program slice (rungs 8, 9, 10) is sufficient to represent the complete behaviour of the program with regard to property 1. Although the upstream valve and the timer are triggered by `L1_PROD` and its value is computed elsewhere in the program, the result is not compromised. Our reasoning is that the property does not attempt to answer the question 'when is `L1_PROD` $= 1$ and therefore `L1_UP` $= 1$' but it actually states 'if such a situation (`L1_PROD` $= 1$) occurs then something else (`L1_UP` $= 1$) will follow'.

C. Properties 2-4

The verification of these properties demonstrates that they have two things in common: none of them can be verified on the model of the program alone (verification is not feasible because of the state explosion), and all of them can be verified using the same level of abstraction.

1) Abstraction: Unlike property 1, these properties (Table I) are not timed. This proves a huge advantage because timers always increase the size of the state space by a considerable margin. Let us look at rungs 9 and 10 first (noting that the same applies to 12 and 13). These two rungs measure the time for which `L1_PROD` $= 1$; and if the time reaches 5 seconds then `L1_PUMP` and `L1_DOWN` are both set to 1, meaning that pump 1 is started and its downstream valve is opened. Whenever `L1_PROD` $= 0$, or it is on for less than 5 seconds, both outputs will be set to 0.

What if the outputs were activated by `L1_PROD` $= 1$ and deactivated by `L1_PROD` $= 0$ only without imposing the 5 seconds delay? Well, the program will essentially behave in the same way but there will be no delays caused by the timer. The absence of timers will help to save a lot of state space and improve the feasibility of verification. To achieve the same effect on our program we replaced the *examine if closed* instruction at memory address `TON_L1/DN` with the same instruction at memory address `L1_PROD` in rung 10 and removed rung 9 altogether. Taking the same action for rungs 12 and 13, using appropriate addresses, the resulting model and state space used by verification is much smaller. After applying the above mentioned abstraction the verification of all three properties becomes feasible.

D. Properties in Uppaal specification language

Our model is using the direct address of each variable rather than its alias. This means that `L1_UP` is in fact an integer variable `O_2_0_3` in accordance with its PLC address `O:2/3` (meaning input 3 on digital output card in slot 2). The time scale used is one tick (of a clock) equal to 1 milisecond, i.e. 20 msec. = 20 and 5 sec. = 5000.

All properties have to be tested after the outputs activation but before the inputs scan (Fig. 1). This is important simply because checking values of variables during the program execution does not make any sense (since the control system behaviour is based on the complete execution of the program). Hence, the model of the program contains a state `testing` which is placed after the actual model of the program. All properties are then checked only if the model of the program is in this state. Access to local variables and states in Uppaal specification language is via 'dot notation' like in Java programming language. In order to verify a property while process `P1` is in state `testing`, the definition `P1.testing` has to be part of its definition. The following are the properties in Uppaal specification language that were verified (true) in this paper.

Property 1
```
A[] not (P1.testing and P1.c_O_2_0_3<5000
and P1.O_2_0_2 == 1)
```
Property 2
```
A[] not (P9.testing and P9.O_2_0_3 == 1
and P9.O_2_0_2 == 1 and P9.O_2_0_1 ==1 and
P9.O_2_0_6 == 1 and P9.O_2_0_5 == 1 and
P9.O_2_0_4 ==1)
```
Property 3
```
A[] (P9.testing and P9.I_1_0_5 == 1) imply
(P9.O_2_0_2 == 0 and P9.O_2_0_3 == 0 and
P9.O_2_0_1 == 0 and P9.O_2_0_5 == 0 and
P9.O_2_0_6 == 0 and P9.O_2_0_4 == 0 and
P9.O_2_0_7 == 0 and P9.O_2_0_0 == 0)
```
Property 4
```
A[] (P9.testing and P9.I_1_0_2 == 1) imply
(P9.O_2_0_2 == 0 and P9.O_2_0_3 == 0 and
P9.O_2_0_1 == 0)
```

E. Model size and verification time

This section contains more detailed look at the size of models (table III) and at the verification time (table IV). Model $\mathcal{M}_1$ is the model generated by our Checker tool when the complete ladder logic program was taken as the input. Models $\mathcal{M}_2$ and $\mathcal{M}_3$ were generated using algorithms explained in sections VI-B and VI-C respectively.

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$
Processes	10	2	10
Global variables	11	3	11
Global clocks	1	1	1
Local variables	34	8	28
Local clocks	2	2	0
States	63	12	60
Transitions	116	19	108

TABLE III

SIZES OF TIMED AUTOMATA MODELS

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$
Property 1	-	0:01	-
Property 2	∞	-	1:08
Property 3	∞	-	1:32
Property 4	∞	-	1:16

TABLE IV

PROPERTY VERIFICATION TIME (MIN:S) FOR EACH MODEL.

The linux machine used for verification was Athlon 1.6 GHz with 2GB of RAM.

VII. CONCLUSIONS

This paper presents an approach to automatic verification of control programs. We have defined algorithms that allow a model of the control program to be automatically built and then verified by the model checker Uppaal. We have demonstrated the feasibility of our method by performing a previously studied and manually verified ladder logic program [4], arriving at the first *automatic* verification of *timed properties* for this program. So far we have focused on a subset of ladder logic instructions but the methodology applies more generally. Currently we support seven ladder logic instructions including an on-delay timer. Additional instructions are being added when needed, as was the case with the *one-shot rising* instruction for the case study described in this paper.

The key difficulty of automatic verification is state space explosion, made worse in the context of control programs due to the need to represent timing. This calls for sophisticated abstraction to be incorporated in the translation recipes. Indeed, none of the properties of the case study were feasible for the detailed, non-reduced model. Although the capability of the Uppaal model checker is likely to continue to improve, it is fair to say that the potential of our method lies in the domain of small but non-trivial, critical components, whose reliability is crucial to the safe running of the control system.

Automatic verification is sometimes referred to as "push button technology", meaning that no interaction, except pushing a button, is needed from the person doing verification. In an ideal world, one would simply input a control program and requirements, and expect verification results to be produced without human interaction. This paper constitutes work in progress towards this goal. We have experimented with various abstraction methods, with some performed by hand before implementing them within the tool, concentrating on the model building and transformation recipes. We believe further efficiency improvements are possible; this, as well as more abstraction methods and evaluation through examples, is what we plan to research next.

Another area for further development is interaction with users, particularly concerning the requirements and back-translation of results of the verification which should be annotated in terms of the ladder logic, rather than the model. The main reason for this is that we are not yet sure about the choice of the requirements language; it should be powerful yet easy for control system programmers to learn and use.

REFERENCES

[1] The International Electrotechnical Committee, "IEC 61131-3, Programmable controllers, *Programming languages*," March 1993.

[2] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994, available at http://www.cis.upenn.edu/~alur/pub.html.

[3] Uppaal model checker homepage http://www.uppaal.com.

[4] Jean-Marc Roussel and Bruno Denis, "Safety Properties Verification of Ladder Diagram Programs," *Journal Européen des systèmes automatisés*, vol. 36(7), pp. 905–917, 2002.

[5] G. Frey and L. Litz, "Formal methods in PLC programming," in *International Conference on Systems, Man, and Cybernetics, Nashville, Tennessee, USA*, October 2000, available at http://www.eit.uni-kl.de/litz/members/frey/PDF/V132.pdf.

[6] A. Mader and H. Wupper, "Timed Automaton Models for Simple Programmable Logic Controllers," in *Proceedings of the 11th Euromicro Conference on Real Time Systems*. IEEE Computer Society, 1999, pp. 114–122, also available at http://www.cs.kun.nl/~mader/papers.html.

[7] Henning Dierks, "PLC–Automata: A New Class of Implementable Real–Time Automata," in *Transformation-Based Reactive Systems Development (ARTS'97)*, ser. Lecture Notes in Computer Science, M. Bertran and T. Rus, Eds., vol. 1231. Springer-Verlag, 1997, pp. 111–125, also available at http://semantik.informatik.uni-oldenburg.de/persons/henning.dierks.

[8] ——, "Specification and Verification of Polling Real–Time Systems," Ph.D. dissertation, University of Oldenburg, July 19, 1999.

[9] R. Huuck, "Transformation of Timed Condition/Event Systems into Timed Automata: An Approach to Automatic Verification," Master's thesis, Chair of Software Technology, Christian–Albrecht–University of Kiel, Germany, 1998.

[10] B. Lukoschus, "Composition and Verification of Condition/Event Systems," Chair of Software Technology, Institute of Computer Science and Applied Mathematics, Christian Albrechts University of Kiel, Germany, Tech. Rep., May 1999.

[11] Moby/PLC tool homepage, http://theoretica.informatik.uni-oldenburg.de/~moby.

[12] Kronos homepage http://www-verimag.imag.fr/TEMPORISE/kronos.

[13] VERDICT tool homepage http://astwww.chemietechnik.uni-dortmund.de/~verdict/.

[14] VHS project homepage http://www-verimag.imag.fr/VHS/main.html.

[15] A. Mader, H. Wupper, and N. Bauer, "Design of a PLC Program for VHS Case Study 1," University of Nijmegen, Tech. Rep., June 28, 1999, also available at http://www.cs.kun.nl/~mader/papers.html.

[16] K. Kristoffersen, K. Larsen, P. Pettersson, and C. Weise, "Experimental Batch Plant CS1 using Timed Automata and UPPAAL," BRICS, Institute of Computer Science, Aalborg University, Denmark, Tech. Rep., May 6, 1999.