Ladder Metamodeling and PLC Program Validation through Time Petri Nets*

Darlam Fabio Bender[1,2], Benoît Combemale[1], Xavier Crégut[1], Jean Marie Farines[2], Bernard Berthomieu[3], and François Vernadat[3]

[1] Institut de Recherche en Informatique de Toulouse (CNRS UMR 5505)
Université de Toulouse. France
first_name.last_name@enseeiht.fr

[2] Departamento de Automação e Sistemas
Federal University of Santa Catarina. Florianopolis, Brazil
last_name@das.ufsc.br

[3] Laboratoire d'Analyse et d'Archicteture des Systemes (CNRS)
Université de Toulouse. France
last_name@laas.fr

Abstract. Ladder Diagram (LD) is the most used programming language for Programmable Logical Controllers (PLCs). A PLC is a special purpose industrial computer used to automate industrial processes. Bugs in LD programs are very costly and sometimes are even a threat to human safety. We propose a model driven approach for formal verification of LD programs through model-checking. We provide a metamodel for a subset of the LD language. We define a time Petri net (TPN) semantics for LD programs through an ATL model transformation. Finally, we automatically generate behavioral properties over the LD models as LTL formulae which are then checked over the generated TPN using the model-checkers available in the Tina toolkit. We focus on race condition detection.

1 Introduction

Actually, verifications of Programmable Logical Controllers (PLCs) programs are made through exhaustive testing. This method takes a long time to be executed and some errors may pass unnoticed in complex systems.

Ladder Diagram (LD) programs are difficult to debug and modify because its graphical representation of switching logic obscures the sequential, state-dependent logic inherent in the program design [1]. Not found bugs in PLC programs are often very costly and sometimes are even a threat to human safety. This work aims to provide a framework for automatic formal verification of PLC programs written in LD language.

To perform the formal verification of LD programs we introduce a model driven approach. LD models are designed according to an LD metamodel. These LD models are then translated into a formal language. The generic LTL properties to be verified are

* This work is supported by the TOPCASED project, part of the french cluster *Aerospace Valley* (granted by the french DGE), cf. http://www.topcased.org

generated automatically, and finally we rely on model-checking to verify the temporal properties. For this work we have chosen the time Petri nets (*TPN*) as formal language, and the *Tina*[1] toolkit for simulation and model-checking. We focus on the verification of generic, i.e. model independent, properties, especially the absence of race conditions in LD programs.

This paper is organized as follows. Section 2 introduces the PLCs, the LD language and the validation issue. Section 3 presents the approach used to formally verify LD programs. Section 4 presents the related work and Section 5 concludes and presents some perspectives and future work.

2 PLCs, Ladder Diagrams and Validation Issue

2.1 Programmable Logical Controllers (*PLCs*)

A PLC is a special purpose industrial computer used to automate industrial processes. It can be connected to several inputs and outputs, and can be programmed to control the state of the outputs depending on the configuration of the inputs and its internal state.

The PLC execution follows a cycle. The state of all inputs is copied into memory. Then, the internal program runs and creates in memory a temporary table of all outputs. When this program completes, the table is written to the outputs and a new cycle starts. It repeats as long as the PLC is running.

A PLC can be programmed using any of the five languages [2] which are: Instruction List (*IL*), Structured Text (*ST*), Function Blocks Diagrams (*FBD*), Sequential Function Chart (*SFC*) and Ladder Diagram (*LD*). The semantics of these languages is not strictly defined, certain definitions are missing or contain ambiguities. Some research work has been made to solve these ambiguities, e.g. [3]. This article focuses on the LD language, but the same approach could be used for the other languages.

2.2 Ladder Diagram (*LD*)

Ladder Diagram is the most used language for programming PLCs. It is a graphical language where the basic elements are based on an analogy to physical relay diagrams [4], so it does not represent a big paradigm shift for technicians that are not used to the new computer techniques and programming languages. In Figure 1 is represented a simple LD program in the textual concrete syntax (ASCII art) normally used. We can look at it as a relay diagram where the power flows from the left to the right, passing through the inputs to activate the outputs. The program is delimited by a vertical line on the left representing the *hot* wire, and another one on the right representing the neutral wire. These vertical lines are called the left and right rails.

The horizontal lines (rungs) and the associated elements represent boolean equations. The dependent element of the equation (coil) is represented by the symbol "( )", while the independent elements (contacts) are represented by "| |". A diagonal line is placed in the middle of symbols as in "|/|" to indicate that the negated value of the variable is used. Variables placed in series represent the *AND* boolean function while variables

[1] `http://www.laas.fr/tina/`

```
|        A          B         C       |
+--------| |----+---|/|-------( )-----+
|               |                     |
|        C      |                     |
+--------| |----+                     |
|                                     |
|        C                    D       |
+--------| |------------------( )-----+
```

Fig. 1. Simple Ladder Diagram example

placed in parallel represent the *OR* boolean function. The rungs are executed in order from the top to the bottom. So the program in Figure 1 represents the boolean equations $C = (A \vee C) \wedge \neg B$ and $D = C$. A variable can be used as a coil and as a contact in the same rung, that is the case of variable C in the first equation.

The elements presented so far are the basic elements of LD. A program may also contain more complex elements, the functions and function blocks (*FB*). Functions always produce the same outputs when provided with the same inputs while *FBs* keep an internal state that also influences the outputs. They are represented by a rectangular box with their names inside, their inputs on the left side, and their outputs on the right side. These boxes are connected in the diagram's rungs through its inputs and outputs. The LD language also provides other imperative elements (procedures, goto) that are not treated in the scope of this work.

It can be noted that LD programs are hard to visualize and become unreadable even for small to medium size programs. The validation of these programs through exhaustive testing are very expensive and unsure. A method for their formal verification is then of great help for system engineers.

2.3 Ladder Diagram Validation

In this work we intend to create a framework for automatic verification of temporal properties in LD programs. The properties to be verified over an LD program could be generic (and apply to every model) or specific to one model. Model related properties must be formulated by the programmer, while generic properties only rely on the metamodel concepts and can be automatically generated from the model.

One of the important generic properties to be verified on an LD program is the absence of race conditions [5]. A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time but, because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly. A race condition occurs in an LD program when under fixed inputs and function block states, one or more outputs keep changing their values. In the LD example of Figure 2, the variable A is an input, C and D are memory, B and E are outputs. It can be seen that even if A is kept stable, the variables C, D and E will not stabilize thus it is an example of race condition. This kind of problem is sometimes difficult to detect with traditional techniques, and bugs not detected during the test period can be very costly to correct later. In section 3.4 we present how to express this property as a LTL formulae and how to automatically generate and validate the formulae from the model.

3 MDE-Based Approach for Formal LD Program Validation

The approach we chose to verify properties in LD programs consists in translating the semantics of LD programs to a language formally (mathematically) defined. The chosen formal language is the Time Petri Net (*TPN*) [6].

To define and execute the translation we used a model driven engineering (MDE) approach. LD Programs are models that must conform to the LD metamodel. These models are then translated to a TPN model that conforms to a TPN metamodel. This TPN model is then translated to the input format of the *Tina* tool. General properties over the Ladder model are also automatically generated as LTL formulae. Finally, we use model-checking to verify the properties represented in LTL formulae over the generated TPN. A general schema of the process can be seen in Figure 3. All translations between models cited above were written in ATL (ATLAS Transformation Language) [7]. This language can be used to write translations of type model-to-model (M2M) and model-to-text (M2T). In this case study we have developed one M2M (Ladder2PetriNet) and two M2T (PetriNet2Tina and Properties) transformations.

3.1 Ladder Diagram Metamodel

The first contribution of this study is the definition of a metamodel for a subset of the LD language. An incremental approach has been chosen to build the metamodel. As a first step it was built a metamodel able to represent a significant subset of the LD language. This metamodel can be seen in Figure 4. An LD program (*Program*) is composed of variables (*Variable*) and rungs (*Rung*). A variable may represent an input, output or memory location (*InOutKind*). A rung is composed of elements (*Element*) that denote basic (*BasicElement*) and complex (*ComplexElement*) elements.

A basic element can be "normal_open" or "normal_closed" (*PlacementKind*). It denotes either a coil (*Coil*) or a contact (*Contact*) and always references a *Variable*. Complex elements represent functions and FBs. The attribute *kind* contains the name of the represented function or FB (*Functions*). Complex elements are composed of inputs

```
|        A                  B     |
+--------| |-----------------( )-----+
|        C                  D     |
+--------|/|-----------------( )-----+
|        D                  C     |
+--------| |-----------------( )-----+
|        C                  E     |
+--------| |-----------------( )-----+
```

Fig. 2. Simple example of races in LD

(*Input*), outputs (*Output*) and internal variables (*InternalVariable*). The enumeration *Functions* can be easily extended to contain all the standard functions and FBs.

We introduce in this work the concept of path (*Path*). A rung is composed of paths. A path is a closed circuit by which the power can flow through the contacts to activate a coil. In the example of Figure 1 we have two paths in the first rung, one representing the boolean equation $C = A \wedge \neg B$ and the other representing the equation $C = C \wedge \neg B$. Note that this is a simple decomposition of the original equation. We have a single path in the second rung representing the equation $D = C$.

A path may have any number of contacts or outputs of complex elements as operators (*Operator*) but only one coil or input of complex elements as result (*Result*). The restriction of only one result per path was introduced to facilitate the translation later, but it does not introduce loose of generality because a path with n results can always be decomposed into n paths with one result. Note that *Result* is a generalization of the elements that may have their state changed by the power flow, while *Operator* is a generalization of the elements that are conditions for the power flow.

The data type of an internal variable of complex elements are restricted in this work to *Boolean* and *Integer* (*VariableType*) and the global variables of the program (*Variable*) are restricted to *Boolean*.

3.2 Time Petri Nets, SE-LTL and Tina Toolbox

In this study, we have chosen the technical space of time Petri nets as the target representation for formally expressing our LD semantics. We also have chosen to express our temporal properties as LTL formulae (*Linear Temporal Logic*) over the Petri net associated to a LD model. Then we manipulate Petri nets and LTL formulae within the *Tina* toolkit.

Time Petri Nets. Time Petri net (or TPN) [6] is one of the most widely used model for the specification and verification of real-time systems. TPNs are Petri nets in which a nonnegative real interval $I_s(t)$, with rational end-points, is associated with each transition t of the net [6].

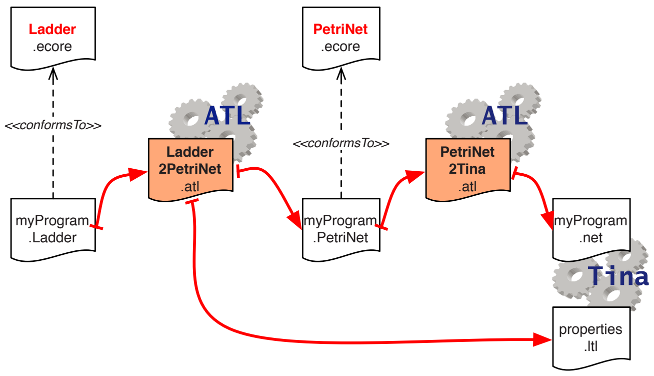

Fig. 3. General overview of the LD validation approach

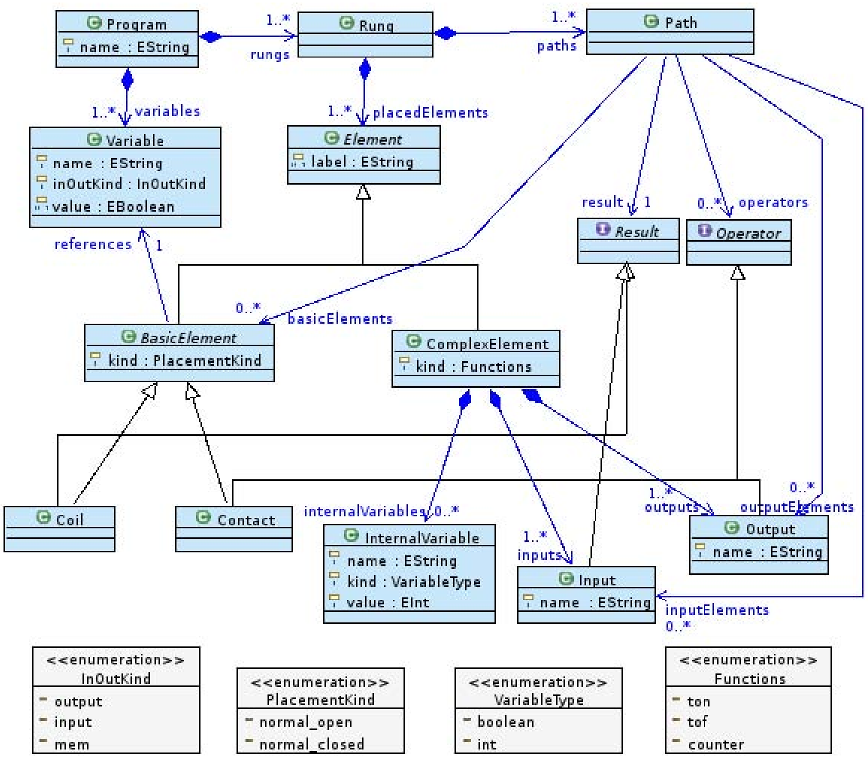

Fig. 4. A simplified LD Metamodel

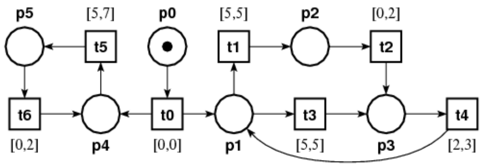

Fig. 5. A Time Petri net

Definition 1. *A* TPN *is a tuple* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$*, in which* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ *is a Petri net, and* $I_s : T \rightarrow \mathbf{I}^+$ *is the* Static Interval *function.*

P is the set of *places*, T is the set of *transitions*, **Pre**, **Post** : $T \rightarrow P \rightarrow \mathbf{N}^+$ are the *precondition* and *postcondition* functions, $m^0 : P \rightarrow \mathbf{N}^+$ is the *initial marking*. $\mathbf{I}^+$ is the set of nonempty real intervals with nonnegative rational end-points.
A Time Petri net is shown in Figure 5.

Let $\mathbf{R}^+$ be the set of nonnegative reals. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left end-point, and $\uparrow i$ its right end-point (if i bounded) or ∞. For any $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ denotes the interval $\{x - \theta | x \in i \wedge x \geq \theta\}$.

States and the temporal state transition relation $\xrightarrow{t@\theta}$ are defined as follow:

Definition 2. *A* state *of a TPN is a pair $s = (m, I)$ in which m is a marking and I is a function called the* interval *function. Function $I : T \rightarrow \mathbf{I}^+$ associates a temporal interval with every transition enabled at m.*
We write $(m, I) \xrightarrow{t@\theta} (m', I')$ if $\theta \in \mathbf{R}^+$ and:

1. $m \geq \mathbf{Pre}(t) \wedge \theta \geq \downarrow I(t) \wedge (\forall k \in T)(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
2. $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
3. $(\forall k \in T)(m' \geq \mathbf{Pre}(k) \Rightarrow$
   $I'(k) =$ **if** $k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k)$
   then $I(k) \dot{-} \theta$
   else $I_s(k))$

States evolve as follows: assume the current state is $s = (m, I)$, t is enabled at m, and became last enabled at time τ. Then t cannot fire before time $\tau + Min(I(t))$, and must fire no later than $\tau + Max(I(t))$, except if firing another transition before t made t not enabled anymore. Firing transitions takes no time.

TPN Metamodel. The time Petri nets metamodel is given in Figure 6. A Petri net (*PetriNet*) is composed of nodes (*Node*) that denote places (*Place*) or transitions (*Transition*). Nodes are linked together by arcs (*Arc*). Arcs can be normal ones or read-arcs (*ArcKind*). The attribute *weight* specifies the number of tokens consumed in the source place or produced in the target one (in case of a read-arc, it is only used to check whether the source place contains at least the specified number of tokens). Petri nets marking is defined by the *tokens* attribute of *Place*. Finally, a time interval can be expressed on transitions.

Model-Checking. For this study, we use $State/Event - LTL$ [8], a linear time temporal logic supporting both state and transition properties. The modeling framework consists of labeled Kripke structures (the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions.

Formulae Φ of $State/Event - LTL$ are defined according to the following grammar:

$$\Phi ::= p \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc\ \Phi \mid \Box\ \Phi \mid \Diamond\ \Phi \mid \Phi\ U\ \Phi$$

Example of $State/Event - LTL$ formulae :

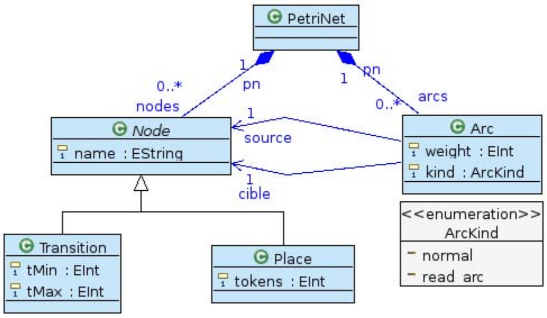

Fig. 6. Time Petri net Metamodel

	(For all paths)
P	P holds at the beginning of the path,
$\bigcirc\, P$	P holds at the next step,
$\Box\, P$	P globally holds,
$\Diamond\, P$	P holds in a future step,
$P\ U\ Q$	P holds until a step is reached where Q holds

Tina Toolbox for Time Petri Nets Verification

*Tina (*TIme Petri Net Analyzer*)* is a software environment to edit and analyze Petri nets and time Petri nets [9]. The different tools constituting the environment can be used alone or together. Some of these tools will be used in this study:

- *nd* (*Net Draw*) : *nd* is an editing tool for automata and TPN, under a textual or graphical form. It integrates a “step by step” simulator (graphical or textual) for the TPN and allows to call other tools without leaving the editor.
- *tina* : this tool builds the state space of a Petri net, timed or not. It can perform classical constructs (marking graphs, covering trees) and also allows abstract state space construction, based on partial order techniques. It proposes, for TPN, all quotient graph constructions discussed in [10].
- *selt*: usually, it is necessary to check more specific properties than the ones dedicated to general accessibility alone, such as boundedness, deadlocks, pseudo liveness and liveness already checked by *tina*. The *selt* tool is a *model-checker* for an enriched version of $State/Event-LTL$. In case of non satisfiability, *selt* is able to build a readable counter-example sequence or in a more compressed form usable by the *Tina* simulator, in order to execute it step by step.
  The *selt* logic is rich enough to encode marking invariants like $\Box\,(p1+p3>=3)$.
  Example of `selt` *formulae* :

$t1 \wedge p2 \geq 2$	for every path $t1$ and $m_0(p2) \geq 2$,
$\Box\,(p2+p4+p5=2)$	a linear invariant marking,
$\Box\,(p2*p4*p5=0)$	a non linear invariant marking,

  selt also allows to define new operators :

$infix\ q\ R\ p = \Box\,(p \Rightarrow \Diamond\, q)$	definition of a “Responds to” operator, (denoted R),
$t1\ R\ t5$	$t1$ “Responds to” $t5$.

3.3 Translational Semantics of LD Programs through Time Petri Nets

In order to apply the model-checking techniques to verify an LD program using the available tools, it is necessary to represent the semantics of the program in a formal language. In this section we describe the translational semantics of an LD program in the form of a TPN.

The semantic transformation is based on the metamodel of the LD language (Figure 4) and the metamodel of the TPN (Figure 6). It was coded using ATL [7]. In this first work we do not translate the complex elements of the LD language, note that the complex elements represent the functions and FB of the LD language, these elements have a unique semantics so a complete translation would have to take into account the specific semantics of each function or FB.

The ATL code is composed of eight matched rules. A matched rule enables to match some of the model elements of a source model, and to generate a number of distinct target model elements. Figure 7 shows what target elements are created for each rules. We use rule inheritance to filter different elements that are represented by a unique class in the metamodel. This inheritance could had been done at the metamodel level but our approach allows to keep the metamodel at a higher abstraction level and also allows to factorize the ATL code.

Each LD Program will be translated to a TPN. This TPN can be divided in three different groups of elements (Figure 8), each one with a different purpose during the simulation.

The first group is responsible for controlling the execution according to the PLC operation, a place is introduced for each execution step, that is, read the inputs, execute all the ordered rungs one by one, then update the outputs. The transitions are timed "[1, 1]" so as to guarantee that all the actions depending on one execution step (these actions are timed "[0, 0]") are executed before the state is changed. This mechanism is used to give a higher priority to transitions associated with the second and third group. We could have achieved the same effect by using prioritized Petri nets (PrTPNs) [11].

The second group represents the value of the variables. Each variable contained in the program will be translated into two places, one representing the False (*VariableName_0*) and the other the True (*VariableName_1*) state of the variable. Transitions are also created to set and reset these variables. Note that transitions of this group – describing the behavior of the environment – evolve in parallel and remain active between the capture of the environment (transition input_reading) and its update (transition reset_variables). This simplified modeling of the environment will result in a very strong combinatorial explosion.

The third group represents the execution semantics. This group is composed of two places for each variable as in the second group, but in this case the variables do not represent the real value of the variable, but are used to calculate the new values. This group represents the simulation variables. Input variables will have their simulation value copied from the real value - on the second group - during the input reading state. Output variables are calculated during the execution of the rungs and at the end, during the output update state, the real value - on the second group - will be updated according to the result of computation. At the end of each cycle all simulation variables are reset.

Figure 7 represents the elements generated for each of the eight rules. In some cases, elements generated by one rule are used by another rule. For example, the place *input_reading* is created by the rule *Program2PetriNet* and then is used by the rule *Input2PetriNet*.

The translation of paths generates the elements that calculate the value of the outputs according to the inputs. In Figure 7 (Path2PetriNet) one can see the elements generated by applying this rule to the first path of Figure 1. This path represents the boolean equation $C = A \wedge \neg B$.

The translation method generates a safe (1-bounded) TPN. This is guaranteed by the transformation. A structural analysis of the TPNs generated by the translation gives one

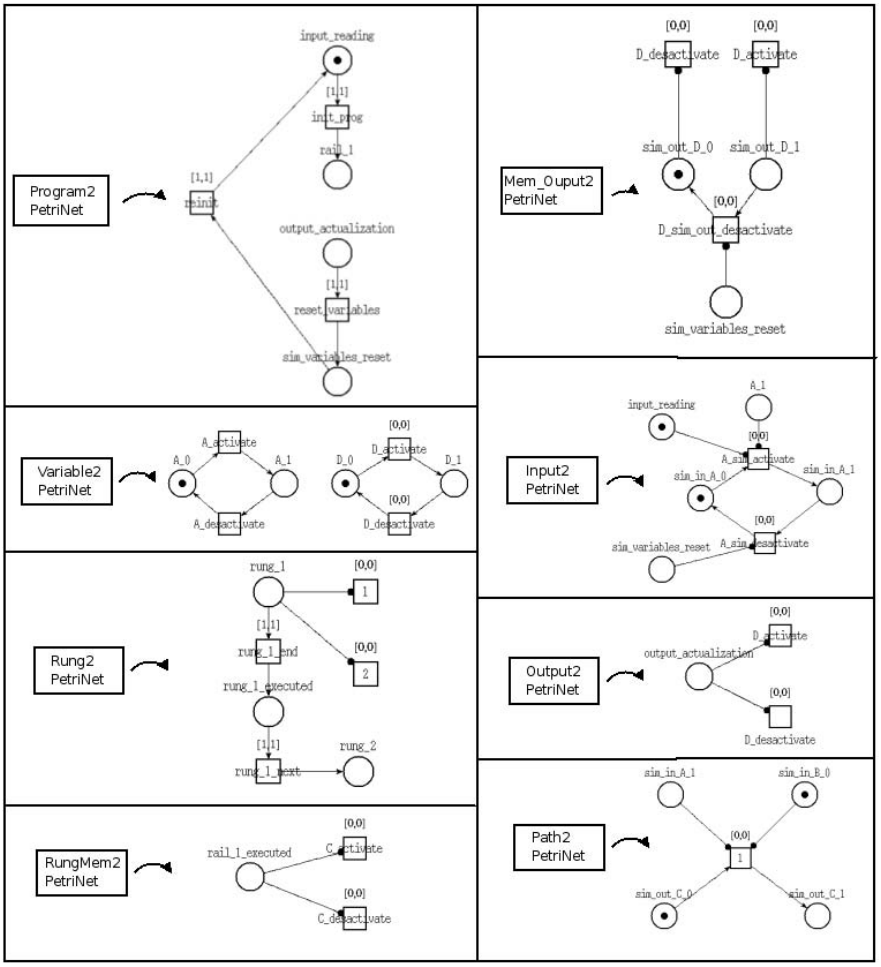

Fig. 7. Schema of the ATL transformation Ladder2PetriNet

invariant for all places in the first group, and one invariant for each couple of places representing the state True or False of variables.

3.4 Formal Verification of Race Conditions in LD Programs

In this section we demonstrate how to express the properties we wish to verify as LTL formulae, how to automatically generate these properties from the LD model and finally how to perform the model-checking with the *selt* tool.

Formalizing Race Conditions as LTL Formulae. To verify the absence of race conditions in an LD program we need to formally describe this property. We first describe the concept of stability of inputs and outputs. A boolean variable is said to be stable if it

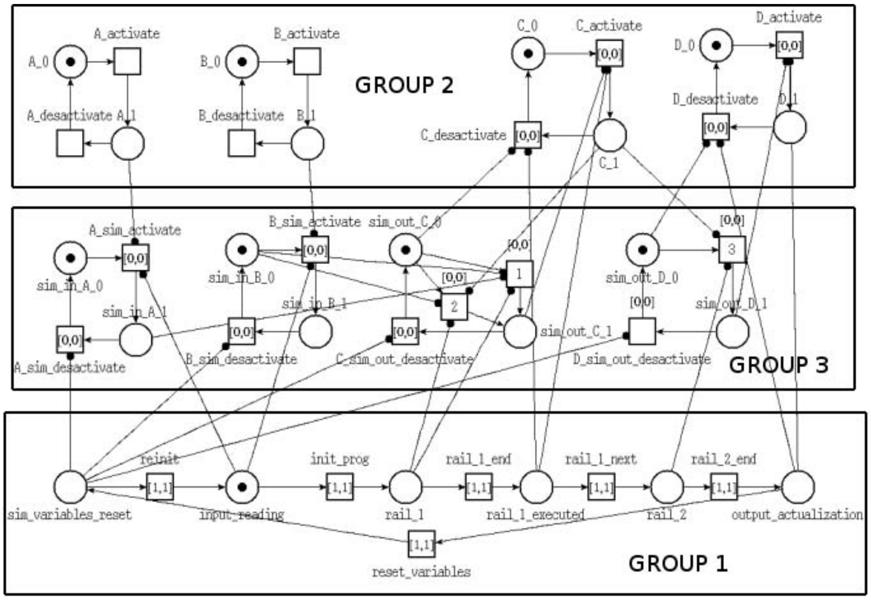

Fig. 8. TPN generated from the LD model on Figure 1

does not change its state, that is, it is always *True* or always *False*. When we translate an LD program, every boolean variable is represented by two complementary TPN places, *(VariableName)_0* representing the *false* state and *(VariableName)_1* representing the *true* state. Based on that we can formulate the Definition 3.

Definition 3. *An LD variable called x is stable if* $((\Box\, x\_0) \vee (\Box\, x\_1))$.

An LD program is free of race conditions if when the inputs are kept stable, all outputs and memory variables will stabilize. This property is represented in Definition 4.

Definition 4. *An LD program is free of race condition if*
$\Box\ (stable\_inputs \Rightarrow \Diamond\ stable\_outputs)$ *where* stable_inputs *represents a logical* AND *between the stability condition for every input variable, while* stable_outputs *represents a logical* AND *between the stability condition for every output and memory variable.*

To reduce the model-checking effort we can decompose the LTL formula into more simple ones. The decomposition is based on the following properties:

$$\Diamond((\Box\, P) \wedge (\Box\, Q)) \equiv \Diamond\Box\, P \wedge \Diamond\Box\, Q$$
$$P \Rightarrow (Q \wedge R) \equiv (P \Rightarrow Q) \wedge (P \Rightarrow R)$$

Our general property can be decomposed in several different properties, one for each output or memory variable. Then we can check separately if every output will stabilize when the inputs are kept stable. When this property does not hold for a variable *x*, the LD program has a race on *x*.

The LTL formula can be simplified even more by applying the cone of influence (COI) reduction. The COI reduction is a technique that reduces the size of a model if

Table 1. Size of the TPN and state space generated

Program	TPN Generated	State Space
Figure 1	23 places, 24 transitions	462 places, 1432 transitions
Figure 2	31 places, 31 transitions	486 places, 1094 transitions

the propositional formula in the specification does not depend on all state variables in the structure [12]. The COI of a variable x is all the variables that influence the value of x. On the LD program represented in Figure 2 the COI of the variable B is $\{A\}$, while the COI of variable C is $\{C, D\}$. So we can redefine the Definition 4 as:

Definition 5. *An LD program is free of race condition if for every output or memory variable V the following property holds:* $\Box\ (COI(V)\_stable \Rightarrow \Diamond\ V\_stable)$, *where* $COI(V)\_stable$ *represents the stability condition for every variable in the COI of* V.

Automatic Generation of LTL Properties. An ATL query was developed to automatically generate the LTL properties in a text format. This query is a M2T transformation that was built over the LD metamodel. It takes as input the LD model and generates the file *properties.ltl*. The COI of a variable is calculated through iterative searching in the model. We can use the genererated file to verify temporal properties with the *selt* tool.

Model-checking. To perform the model-checking we first use the *tina* tool to build the state space of the TPN generated by the translation. The *selt* tool is then used to verify the LTL properties over this state space.

3.5 Results

This method was applied to the LD program in Figure 1 and as expected the property was evaluated to *TRUE*. The program is indeed free of race conditions. When applied to the program in Figure 2 the property was evaluated to *TRUE* for variable B, and to *FALSE* for variables C, D and E, proving that this program has a race on these three variables. In Table 1 is presented the size of the TPN and state space generated for both programs.

The results obtained for the two examples prove that the MDE approach can be used for verification of LD programs, transformation may be run on middle to complex LD diagrams. However, this first approach for translation of LD diagrams to Petri Nets can be largely optimised. The use of temporal constraints to simulate the priorities (instead of inhibitor arcs or directly of priorities[2]) results in an important extra cost. Likewise, the modeling of the environment is very simple (the environment continues to evolve in parallel between two captures) inducing a useless combinatorial explosion. With this translation, the Petri net associated with a LD program able to control a system containing 6 actuators (outputs) and 7 sensors (inputs) generates a state space of 7 million states. The "covering step graph" [13] construction of Tina, exploiting "partial order techniques" [14], allows to reduce drastically this explosion and leads to a state space of 40 states. However, a finer modeling of the environment will have to be carried out to take into account significant LD programs.

[2] Since March 2008, Tina 2.9.0 supports the priorities.

4 Related Works

Numerous authors have been working in formal verification of PLC programs. These works are normally based in two different approaches.

The first approach consists in implementing PLC programs directly in a formal language, and then automatically generate the PLC code. [15] introduces a program based on a set of rules to translate TIPN (*Time Interpreted Petri Nets*) control specifications into Ladder Diagrams. In [16], [17] and [18] a set of tools was presented for implementation of PLC programs using SIPN (*Signal Interpreted Petri Nets*), symbolic model-checking was used for validation and verification and the control specification was finally implemented through IL. [19] introduces a formalism based on timed state machines called PLC-automata and a translation of this formalism into ST code.

The second approach consists in translating the existing PLC programs into a formal language and automatically perform the verification. This approach permits the use of formal languages for the verification of PLC programs without changing the programming paradigm. Our work is based in this second approach. Some works using this approach have been developed. [20] and [21] provide a Petri Nets semantics for a subset of the IL language. [22] provides a framework for automatic verification of programs written in IL, a formal semantics (transition system) is presented for a subset of the IL language, which is directly coded into a model-checking tool (Cadence SMV[3]). It is then possible to automatically verify behavioral properties written in LTL. In [23] LD programs are formally represented through a transition system. In [5] a combination of probabilistic testing and program analysis has been used to detect race conditions in relay ladder programs. [24] generally discusses the transformations between NCES (*Net Condition Event Systems*) State Charts and PLC languages. In [25] is presented a method for formal verification of LD programs through a translation to timed automata. They check model related properties, while we focus on generic properties. The original program is abstracted according to these properties before the generation of the timed automata, in this way, for each property to be verified it may be generated a different timed automata. In [26] is discussed semantic issues and a verification approach for SFC and IL programs. It uses a model-checking framework to verify SFC programs and it suggests static analysis techniques, a combination of data flow analysis and abstract interpretation to verify IL programs.

Our work is based on this second approach and is the first to present an MDE approach for PLCs program validation.

5 Conclusion and Future Work

We have introduced in this article a model driven approach for formal verification of LD programs. We have defined a metamodel for a subset of the LD language. LD models are designed according to this metamodel. An LD model is then translated into a formal language (TPN) in the input format of *Tina* toolkit by means of two ATL transformations. The LTL properties to be verified are generated automatically from the model.

[3] `http://www.kenmcmil.com/smv.html`

Finally we utilize model-checking to verify the temporal properties over the generated TPN.

Some things still remain to be done, the metamodel must be completed to represent the complete LD language. The modelling of LD program s must be refined (specially the modeling of the environment) to reduce the combinatorial explosion.

In order to implement a complete translation of the LD concepts, we need a language directly supporting data processing like Fiacre [27]. Fiacre was designed in the framework of the TOPCASED project [28] dealing with model-driven engineering and gathering numerous partners, from both industry and academics. Therefore, Fiacre is designed both as the target language of model transformation engines from various models such as SDL, UML, AADL, and as the source language of compilers into the targeted verification toolboxes, namely CADP and *Tina*.

Model related properties must be formulated by the programmer, as LD programmers are not specialists in formal verification, a mechanism must be used to generate the LTL properties from the user specification. One of these mechanisms is presented in [29] where is presented an approach for generating LTL properties from TOCL ones, a temporal extension of OCL. In [30] is presented a framework for generating formal correctness properties from natural language requirements. The return of the model-checking counter-examples to the user must be implemented in order to hide to the user the complexities of the target language.

We are currently working on the recognition of the textual concrete syntax (ASCII art) specified in the IEC 61131-3 standard. We want to be able to generate LD models from its textual syntax. The metamodel presented in this article tries to represent the LD language in a format closer to its semantics. For the recognition of the textual syntax we should create an intermediate metamodel that is closer to this syntax and then apply a model transformation to obtain our original model. We are studying the possible solutions, one is the use of TCS [31], that relates metamodel concepts with the textual concrete syntax and allows to build a model from the textual syntax (injector) and also to generate concrete syntax fromo a model (extractor).

Our work demonstrates that it is possible to apply a model driven approach in formal verification of PLC programs. The combinatorial state explosion is a recurrent problem in the formal verification community and several works have been developed to overcome this problem.

References

1. Guasch, A., Quevedo, J., Milne, R.: Fault diagnosis for gas turbines based on the control system. Engineering Applications of Artificial Intelligence 13(4), 477–484 (2000)
2. International Electrotechnical Comission: IEC 61131-3 International Standard, Programmable Controllers, Part 3: Programming Languages (2003)
3. Tourlas, K.: An assessment of the IEC 1131 -3 standard on languages for programmable controllers. In: Daniel, P. (ed.) SAFECOMP 1997: the 16th International Conference on Computer Safety, Reliability and Security York, UK, September 7-10, 1997, pp. 210–219. Springer, Heidelberg (1997)
4. Schum, J.L.: Locksmithing and Electronic Security Wiring Diagrams. McGraw-Hill Professional, New York (2002)

5. Aiken, A., Fähndrich, M., Su, Z.: Detecting races in relay ladder logic programs. In: Steffen, B. (ed.) ETAPS 1998 and TACAS 1998. LNCS, vol. 1384, pp. 184–200. Springer, Heidelberg (1998)
6. Merlin, P., Farber, D.: Recoverability of communication protocols–implications of a theoretical study. Communications, IEEE Transactions on [legacy, pre - 1988] 24(9), 1036–1043 (1976)
7. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
8. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
9. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research 42(14), 2741–2756 (2004)
10. Berthomieu, B., Vernadat, F.: Time petri nets analysis with tina. In: Third International Conference on Quantitative Evaluation of Systems, 2006. QEST 2006, pp. 123–124 (2006)
11. Berthomieu, B., Peres, F., Vernadat, F.: Model-checking bounded prioritrized time petri nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 516–535. Springer, Heidelberg (2007)
12. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
13. Vernadat, F., Azéma, P., Michel, F.: Covering step graph. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 516–535. Springer, Heidelberg (1996)
14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
15. Jimenez, I., Lopez, E., Ramirez, A.: Synthesis of ladder diagrams from petri nets controller models. In: Proceedings of the 2001 IEEE International Symposium on Intelligent Control, 2001 (ISIC 2001), pp. 225–230 (2001)
16. Minas, M., Frey, G.: Visual plc-programming using signal interpreted petri nets. In: American Control Conference, 2002. Proceedings of the 2002, vol. 6, pp. 5019–5024 (2002)
17. Klein, S., Frey, G., Litz, L.: A petri net based approach to the development of correct logic controllers. In: Proceedings of the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT 2002), Grenoble (France), pp. 116–129 (2002)
18. Frey, G.: Design and formal Analysis of Petri Net based Logic Control Algorithms (Dissertation, University of Kaiserslautern). Shaker Verlag, Aachen (2002)
19. Dierks, H.: PLC-automata: a new class of implementable real-time automata. Theoretical Computer Science 253(1), 61–93 (2001)
20. Heiner, M., Menzel, T.: Instruction list verification using a petri net semantics (1998)
21. Heiner, M., Menzel, T.: A petri net semantics for the plc language instruction list. In: IEE workshop on discrete event systems (1998)
22. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of plc programs written in instruction list. In: 2000 IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2449–2454 (2000)
23. Moon, I.: Modeling programmable logic controllers for logic verification. Control Systems Magazine, IEEE 14(2), 53–59 (1994)
24. Rausch, M., Krogh, B.: Transformations between different model forms in discrete event systems. In: Computational Cybernetics and Simulation, 1997 IEEE International Conference on Systems, Man, and Cybernetics, 1997, October 12-15, 1997, vol. 3, pp. 2841–2846 (1997)

25. Bohumir Zoubek, J.M.R., Kwiatkowska, M.: Towards automatic verification of ladder logic programs. In: Proc. IMACS Multiconference on Computational Engineering in Systems Applications (CESA) (2003)
26. Huuck, R.: Software Verification for Programmable Logic Controllers. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel (2003)
27. Berthomieu, B., Farail, P., Gaufillet, P., Peres, F., Bodeveix, J.P., Filali, M., Saad, R., Vernadat, F., Garavel, H., Lang, F.: FIACRE: an intermediate language for model verification in the TOPCASED environment. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse SEE (electronic medium) (2008), `http://www.see.asso.fr`
28. Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J.P., Chemouil, D.: The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development. In: Data Systems In Aerospace (DASIA), Berlin, Germany, 22/05/2006-25/05/2006, European Space Agency (ESA Publications) (2006), `http://www.esa.int/publications` (electronic medium)
29. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X., Vernadat, F.: A Property-Driven Approach to Formal Verification of Process Models. In: Cardoso, J., Cordeiro, J., Filipe, J., Pedrosa, V. (eds.) Enterprise Information System IX. Springer, Heidelberg (2008)
30. Nikora, A.P.: Developing formal correctness properties from natural language requirements. NASA: Jet Propulsion Laboratory (2006)
31. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: 5th international conference on Generative Programming and Component Engineering (GPCE 2006) (October 2006)