Edola: A Domain Modeling and Verification Language for PLC Systems

Hehua Zhang
School of Software, KLISS, TNLIST
Tsinghua University
Beijing, China
Email: zhanghehua@gmail.com

Ming Gu
School of Software, KLISS, TNLIST
Tsinghua University
Beijing, China
Email: guming@tsinghua.edu.cn

Xiaoyu Song
Dept. ECE
Portland State University
Oregon, USA
Email: song@ee.pdx.edu

***Abstract*—Formal modeling and verification of PLC systems become paramount in engineering applications. The paper presents a novel PLC domain-specific modeling language Edola. Important characteristics of PLC embedded systems, such as reactivity, scan cycling, real-time and property patterns, are embodied in the language design. Formal verification methods, such as model checking and automatic theorem proving, are supported in Edola modeling. The TLA$^+$ specification language constitutes an intermediate language layer between Edola and the verification tools, enhancing a large degree of reusability. A prototype IDE for Edola and its seamless integration of a model checker TLC and an automatic theorem prover Spass are implemented. A case study illustrates and validates the applicability of the language.**

Keywords-domain-specific modeling language; formal verification; PLC; TLA+.

I. INTRODUCTION

Programmable Logic Controllers (PLCs) are widely used in industry for embedded systems [1]. A PLC interacts with its environment, following a so-called scan cycling mechanism. It starts with inputting environmental data, then performs a local computation, and finally outputs the results to the environment [2]. With their increasing use, PLC systems become more and more complex. Formal modeling and verification becomes paramount in PLC engineering applications to ensure the correctness.

There are several expressive formal modeling languages that has been adopted in the modeling and verification of PLC systems, such as timed automata [3], timed Petri net [4], SMV [5] and TLA$^+$ [6].

Many PLC modeling work focus on a high level of abstraction, so that a small model can be obtained for verification. However, the characteristics like scan cycling are not considered, and a wide gap exists between the abstract models and their PLC implementations. To get a suitable level of abstraction to model PLC systems, their characteristics like reactivity, scan cycling, real-time and property patterns should be embodied in the modeling language. Although the existent formal modeling languages are powerful, the characteristics of PLC applications are not directly supported.

In this paper, we presented a novel PLC domain-specific modeling language Edola, which provides notations for better understanding and easier modeling of applications in the PLC domain. Formal verification methods, like model checking and automatic theorem proving are supported in Edola modeling. Edola provides a suitable level of abstraction to model PLC systems, which can express features of PLC systems and also rule out unnecessary details. We adopt the TLA$^+$ specification language as an intermediate layer between the Edola language and the verification tools, to enhance a large degree of reusability. With the inherent logic of TLA$^+$, it is possible to verify an Edola model with a state based method like model checking and also a logic based reasoning method like theorem proving. A prototype IDE for Edola has been implemented, which provides both the user-interface for modeling and the seamless integration of two verification tools: TLC [7] and Spass [8].

The paper is organized as follows. We introduce the syntax, the intuitive semantics of Edola in Section III. The formal semantics is illustrated in Section IV. Section V explains the verification method of Edola models, including the transformation rules and the optimization strategies taken in the procedure. Section VI introduces the prototype IDE tool. A case study is illustrated in Section VII to validate the applicability of the language. Finally, we conclude our work in Section VIII.

II. AN EXAMPLE: FIRE-FIGHTING PLC CONTROL SYSTEM

To better explain the language Edola and its tool, we first introduce a fire-fighting PLC control system which is used in ship docks. We will take part of the case now and then, to explain the syntax, semantics and our design considerations of Edola.

This running case is a system used to fight fire that may happen at ship docks. It operates the fire-fighting cannons under the control of a user and displays information about the current operating state. The cannons are used in some specified fire cases and are connected with several valves. When there is a fire-fighting request, the user can control the equipments in the control panel. A possible designed control panel with two cannons and two fire-fighting cases are sketched in Figure 1. The preparatory steps of the operations are as follows: (1) powering up the system; (2)

 ISBN: 978-1-61208-165-6

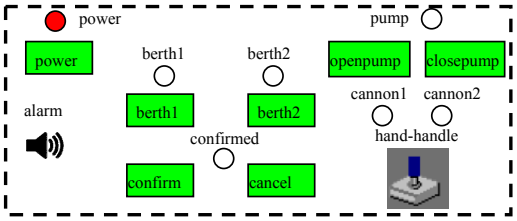

Figure 1. A sketch of the control panel of the system

opening the pump; (3) selecting the firing place (case); (4) confirming the selection; (5) controlling the direction of the cannon by the hand-handle. After the fire fighting, the user proceeds as follows to shut down the system: (6) closing the pump; (7) canceling the current fire-fighting case; (8) restarting the system for another fire-fighting case or powering down the system to finish. Note that because of the technic requirement, only one cannon can be used at the same time. On the other hand, the system should consider the case when the controlled devices or the communication network go wrong, and the users should be alarmed. In this example, we consider a typical alarm, that is, when the PLC program sends the command to open the pump, while the pump does not open in 5 seconds, the beeper in the control panel rings 3 seconds and the control system goes to the initial state.

III. The Syntax of Edola

An outline of the Edola syntax is shown in Figure 2. Details are omitted for the sake of space limitation. The Edola language is composed by modules. The main body includes module extensions, static declarations, dynamic definitions and verification requests. An EXTENDS statement can extend standard modules like Naturals, Reals in Edola or user-defined modules.

The auxiliary symbols are declared by a series of Edola formulas (**GeneralDef**).Taking the fire-fighting case as an example, we can use $Direction == \{$"up", "down", "left", "right", "none"$\}$ to denote the possible moving directions of a cannon, where $Direction$ is defined as a enumeration type with 5 elements.

Constant declarations start with CONSTANT , denoting the parameters of a module. The declared parameters cannot be changed in the latter dynamic definitions.

Variable declarations are classified by the input variables (INPUTVAR), output variables (OUTPUTVAR) and system variables (SYSTEMVAR), according to the PLC scan cycling mechanism. Input variables denote the environment of a PLC software, including the commands from users by the control panels and the signals from the physical devices. The values of input variables are unchanged during a single scan cycle. Output variables denote the output signals of a PLC software, controlling the moving of physical devices or

```
EDOLA-module ::= AtLeast4("-") MODULE ModuleName (AtLeast4("-")
                 ( nil | EXTENDS CommaList( Name) )
                 GeneralDef
                 Declarations
                 ActionDef
                 Constraints
                 (nil | Properties)
                 AtLeast4("=")
GeneralDef ::= nil | ( formula )*
Formula ::= LeftF '= =' Exp
LeftF ::= Name | Name "(" CommaList(ID) ")"
Declarations ::= ConstDeclarations VarDeclarations
ConstDeclarations ::= nil | CONSTANT CommaList(OpDec)
OpDec ::= ConstName | ConstName "(" CommaList("_") ")"
VarDeclarations ::= INPUTVAR varDecList
                    OUTPUTVAR varDecList
                    SYSTEMVAR varDecList
ActionDef ::= INIT formula
              ACTION ActionList
ActionList ::= ( Formula )+
Constraints ::= EnvConstraint
                ( nil | TimeConstraints )
EnvConstraint ::= ENV TOTAL
                | ENV ( Formula )*
TimeConstraints ::= TIME ( Duration | Interval | Delay | Deadline | Timeout | Waituntil ) +
Properties ::= PROP PropName ": "
               ( Respond | Compete | Sequence | Priority | Inv | ActInv )+
Respond ::= (nil | Quantif) RESPOND "(" Actname, SysStateExp, EnvStateExp ")"
Quantif ::= (\A | \E) Name \in SetName
.....
```

Figure 2. The excerpt of the Edola syntax

displaying the status of the system in control panels. System variables are used by the PLC software to implement the controlling functionalities. The values of system variables are usually changed during a scan cycle. The variables are defined by their name and type. For example,

$$\begin{aligned} &\text{INPUTVAR } realPump \in \text{BOOLEAN} \\ &\text{OUTPUTVAR } alarm \in \text{BOOLEAN}, \\ &\qquad s\_handle \in [Cannon{->}Direction] \\ &\text{SYSTEMVAR } state \in SysState \end{aligned} \tag{1}$$

declares an input boolean variable $realPump$, which denotes the opening state of the water pump; an output boolean variable $alarm$ to denote the beeper rings or not , the other output variable s_handle representing the control commands to the water cannons, which is an array represented in the functional style. A system variable $state$ is also declared to represent the current state of the PLC software.

The dynamic definitions describe how the PLC software works in a specified environment. The behaviors of PLC software are defined by an initial state and a series of actions in Edola. The initial state is represented by the keyword INIT

 ISBN: 978-1-61208-165-6

following an Edola formula, to assign the initial value for all the variables. The keyword ACTION starts the definition of a series of actions. Each action is represented by an Edola formula, to define the next-state action of PLC software in the current system state. Edola permits the definition of parameterized actions. For example, the following formula

$$
\begin{aligned}
&SelectCase(i) == \qquad (2)\\
&(\forall j \in UnlockedButton : j\#i \Rightarrow\ u\_button[j])\\
&\ i \in FireCase\\
&\wedge\ u\_button[\text{“power”}]\\
&\wedge\ s\_sysState \in \{\text{“pumpopened”}, \text{“selected”}\}\\
&\wedge\ u\_button[i]\\
&\wedge\ s\_buttonLight' = [j \in ButtonLight \mapsto\\
&\qquad \text{IF } j \notin FireCase \text{ THEN } s\_buttonLight[j] \text{ ELSE}\\
&\qquad \text{IF } j = i \text{ THEN TRUE ELSE FALSE}]\\
&\wedge\ s\_sysState' = \text{“selected”}
\end{aligned}
$$

defines the on-fire case selecting action, which means that, for any fire case, if the power is on, the system is in the expected state, only the button for fire case i is pushed down, the case i is successfully selected. Only the corresponding light for case i is set to on and the system state is modified accordingly. The type of the parameter should be specified in the formula definition, which will be checked by the Edola compiler.

PLC applications are reactive, thus the environment should also be specified besides the behaviors of a PLC software. Edola provides two possibilities for environmental modeling. Users can define the specific behaviors of the environment by a series of formulas starting with the keyword ENV or use the keyword ENV TOTAL to leave the environment modeling work to Edola compiler. In the latter case, the compiler will generate a complete environmental model automatically, which covers all the possibilities of the environmental inputs.

When there are time constraints on system behaviors, they can be described in Edola by the part starting with the keyword TIME . We provide several time operators for describing the constraints on an action (*Duration*) or on the interval between actions (*Interval*), respectively. Four advanced operators *Delay*, *Deadline*, *Timeout* and *WaitUntil* are also supported for the usability. For example, the opening pump time limit 5 can be represented in Edola by applying the *Timeout* operator on the action $OpenPump$ and the action $BeeperRing$: TIMEOUT $(OpenPump, BeeperRing, 5)$.

The verification requests are represented in Edola by a series of properties, and start with the keyword PROP . The given properties should be checked whether they are satisfied by the PLC software behaviors under the specified environment and the requested time constraints. Edola provides six property patterns: the responding properties (with the keyword RESPOND), the competing properties (COMPETE), the sequential properties (SEQUENCE) , the priority properties (PRIORITY) and two patterns more general: state invariants (STATEINV) and action invariants (ACTINV). For example, in a fire-fighting application, the correctness property

$$
\begin{aligned}
&CannonUsedOnlyOne : \qquad (3)\\
&\quad \backslash\text{A } i \backslash in\ Cannon, j \in Cannon :\\
&\quad \text{COMPETE } (i\#j, Selected[i], Selected[j])
\end{aligned}
$$

denotes the competing requests among selection of cannons: at any moment, at most one cannon can be selected. Note that all the provided property patterns are safety properties, which denotes in general that something bad will never happen.

IV. THE FORMAL SEMANTICS OF EDOLA

In this section, we give the formal semantics of Edola by the transformational method with the specification language TLA$^+$.

A. Preliminaries of TLA$^+$

TLA$^+$ [9] is a formal specification language based on the Temporal Logic of Actions TLA, first-order logic and Zermelo-Fränkel set theory. It is un-typed, abstracted and widely used in the high-level specification of concurrent and reactive systems.

The characteristic form of the TLA$^+$ specification of a transition system is a formula of the form $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$, where $vars$ is a tuple containing all state variables of the system. The first conjunct $Init$ describes the possible initial states of the system. The second conjunct of the specification asserts that every step (i.e., every pair of successive states in a system run) either satisfies $Next$ or leaves the term $vars$ (and therefore all state variables) unchanged.The third conjunct L is a temporal formula stating the liveness conditions of the specification, and in particular can be used to rule out infinite stuttering.

B. Module extensions and static declarations

The semantics of an Edola module is given by a TLA$^+$ module. The EXTENDS and CONSTANT statements of Edola are assigned the same semantics with the ones in TLA$^+$. The input variables, output variables and system variables are explained by the variables declarations in TLA$^+$ together with a type invariant to be ensured, since TLA$^+$ is an un-typed language.

C. Dynamic definitions

The definition of the PLC behaviors in Edola includes the INIT part and the ACTION part. The two components are illustrated by the corresponding TLA$^+$ components, intuitively. The INIT part in Edola corresponds the initial state formula in TLA$^+$, which is composed by the whole INIT definition in Edola with the conjunction of the initial value of the variable *aux*, that is, zero. An action definition

in the ACTION part of Edola corresponds an action definition in TLA+, with the conjunction of an UNCHANGED statement.

The value changing of the variables in TLA+ is total, which requests the explicit statements of all the variables that are unchanged. This feature is good for mathematical reasoning, but is unintuitive and tedious for writing a model [10]. Edola then possesses the advantages on both modeling and mathematical reasoning by the transformational semantics. The semantics of the complete Edola specification (based on TLA+) depends on the time description. We will explain it in the end of this section.

The semantics of the environmental model lies on whether it's a total one generated by the Edola compiler or not. The former is illustrated by an TLA+ action *EnvInput* which changes the input variables randomly on condition of the type invariant is satisfied. The latter corresponds to the TLA+ action definitions, similar to the ones for PLC actions , except that the UNCHANGED statement denoting the unchanged value for the input variables that are not defined instead. The transformation for ENV TOTAL is shown in Table I.

Table I
THE FORMAL SEMANTICS OF ENV TOTAL IN EDOLA

Edola definitions	The formal semantics with TLA+
ENV TOTAL	$EnvInput \triangleq \bigwedge_{i \in n_i} invar_i' \in ValRange_i$

A real-time TLA+ module *RealTimeNew* is provided to interpret the time operators in Edola. The time in Edola is logical and continuous, which was interpreted with a real type variable *now* in TLA+. A time constraint in Edola is then interpreted on an action denoting the constraints of its enabling time with the running of real time *now*. Each time pattern in Edola corresponds a defined time action in *RealTimeNew*. The details of the time module are introduced in our work [11].

Finally, we define the formal semantics of the Edola dynamic behaviors. When there isn't any time constraint in a Edola module, the whole Edola specification is interpreted as the formula $SpecName \triangleq Init \wedge \Box[Next]_{vars}$, where $SpecName$ is needed in TLA+, so it is generated by the Edola compiler according to the module name. $Init$ is the formula name used for defining the initial state in Edola and *Next* is defined by

$$\begin{aligned} Next \triangleq \; & \vee \wedge aux = 0 \\ & \quad \wedge EnvInput \wedge \text{UNCHANGED } SOV \\ & \quad \wedge aux' = 1 \\ & \vee \wedge aux = 1 \\ & \quad \wedge SystemAction \wedge \text{UNCHANGED } IV \\ & \quad \wedge aux' = 0. \end{aligned} \tag{4}$$

EnvInput denotes the environment model we introduced above. *SOV* denotes the system variables and the output variables declared in Edola, while *IV* denotes the declared input variables. *SystemAction* defines a complete set of possibilities for PLC responses, which includes all the expected actions defined in the ACTION part, and the case when none of them are enabled.

When there are time constraints in an Edola module, the Edola specification is interpreted in a module $RTModule$, which extends a $FuncModule$ for the functional modeling. The specification is then illustrated by $SpecName \triangleq BigInit \wedge \Box[BigNext]_{RTvars} \wedge RTL$, where the functional semantic interpretations of Edola like $Init$, $Next$ are same as the former introduced ones, but encapsulated in the module $FuncModule$. The initial state $BigInit$ and the next-state action $BigNext$ of the timed specification is composed by the functional parts and the settings of the variable *now* and the n timers $t_1, \ldots, t_n$ appeared in the TIME part of the Edola model.

D. Verification requests

The property definitions provided in Edola are illustrated by the property definitions with temporal logic of actions (TLA) in TLA+ language. The excerpt of the translation is shown in Table II.

Table II
THE FORMAL SEMANTICS OF PROPERTY DEFINITIONS IN EDOLA

Edola Property definitions	The formal semantics with TLA+
RESPOND $(Act, EnvS, SysS)$	$\Box(EnvS \wedge \neg SysS \Rightarrow \neg(\text{ENABLED } Act))$
COMPETE $(Cond, S_1, S_2)$	$\Box(Cond \Rightarrow \neg(State_1 \wedge State_2))$
SEQUENCE $(Act, SysS)$	$\Box[Act \Rightarrow SysS]_{vars}$
PRIORITY $(Act, SysS)$	$\Box(SysS \Rightarrow (\text{ENABLED } Act \wedge \neg \text{ENABLED } (OtherActs)))$
STATEINV $(SysS)$	$\Box(SysS)$
ACTINV (Act)	$\Box(Act)_{vars}$

V. THE VERIFICATION OF EDOLA MODELS

Providing the automatic verification support for the Edola language is important to improve its usability. Model checking and automatic theorem proving are the two dominant automatic verification methods. The Edola compiler implements the support for both model checking and automatic theorem proving, with the intermediate language TLA+. To make the verification procedure pragmatically efficient, we took two major optimization strategies in the transformation procedure.

First, when the environmental model is specified by the ENV TOTAL keyword, the compiler will generate the formula for $EnvInput$ with the input clearing action *ClearEnvInput* added, which resets the values of all the input variables to the initial value. According to the PLC scan cycling mechanism, PLC gathers the new values of its environment at the beginning of each cycle, so the values of the input

 ISBN: 978-1-61208-165-6

variables in the last cycle are discarded. It ensures the logical correctness of adding a *ClearEnvInput* action in the end of each cycle. The new definition of $Next$ saves much state space and search space for verification and thus improves the efficiency.

Second, when all the verification requests are functional properties (which means that none of the timer variables appears in the PROP part of Edola), even though some time constraints are described in the TIME part, the compiler will generate a un-timed TLA$^+$ model for it. The reason is that the timed model *RTModule* refines its functional part *FuncModule*: $RTModule \Rightarrow FuncModule$, so that for each property P, if it holds on the functional model, say $FuncModule \Rightarrow P$, with the transitivity of logic implication, it also holds for the timed model $RTModule$. As a result, we can check the functional properties on the functional part instead of the complete one, thus a better space and time cost can be saved in the verification.

The model checking of the Edola model is achieved by the transformation and the TLC model checker. As to the automatic theorem proving support, we can prove inductive invariants with the reasoning rule in TLA$^+$:

$$\frac{Init \Rightarrow P, \quad P \wedge Next \Rightarrow P', \quad P \wedge v = v' \Rightarrow P'}{Init \wedge \Box[Next]_v \Rightarrow \Box P}. \quad (5)$$

The verification is then reduced to the first-order logic level without temporal operators.

VI. The Edola tool

We implemented a prototype IDE to model and verify PLC systems with the Edola language, see Figure 3. The tool includes an editor to write the model, and a compiler to check, transform and verify the model. It is implemented with Java 1.6. The interface is developed with Netbeans IDE 6.7 and the compiler is implemented with the scanner/parser generator JavaCC 4.2 with the JJTree preprocessing functionality. The compiler implements syntax and semantic checking, and then the seamless integration of the model checker TLC and the automatic theorem prover Spass to verify an Edola model.

Beside the general semantic checking same with other language compilers, the Edola compiler provides also semantic checking specific to PLC applications. We check whether the actions defined in ACTION part are possible to execute one by one. If an action can never be executed, an alarm information is provided. We also check whether the disjunction of conditions for all the actions defined in ACTION part is TRUE. If not, an alarm is provided.

The semantic checking and the later model checking and automatic theorem proving provides strong verification of Edola models. The checking procedure is completely automatic.

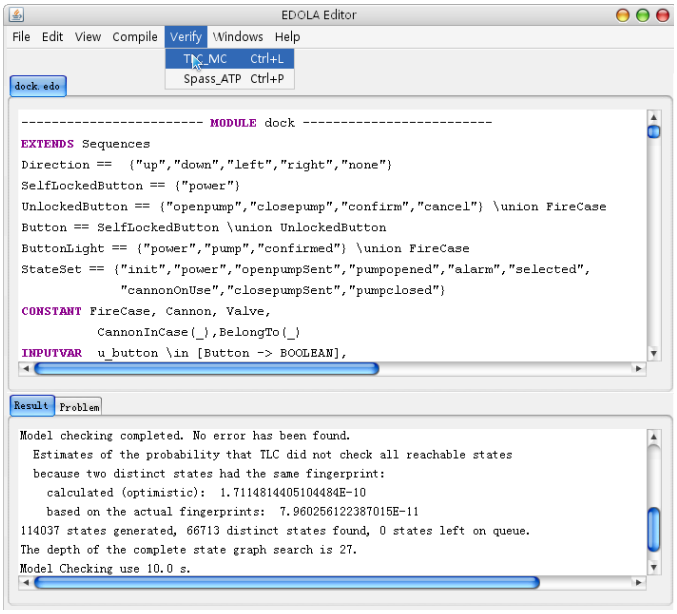

Figure 3. The prototype IDE of Edola

VII. Case study

The Edola language and its IDE has been applied in several medium-scale PLC applications,like the answering machine problem, the steeves control in a theater and the fire-fighting controls in a dock. In this section, the case about a fire-fighting PLC system used for the docks is chosen and presented to further illustrate and validate the Edola language and its tool. We introduce the Edola model, the TLC model checking and the Spass automatic theorem proving of it, respectively.

In the Edola model, we set the physical connections of the fire-fighting system as parameters. They are declared as: $FireCase$ for the set of fire cases, $Cannon$ for the set of used cannons, $Valve$ for the set of valves, $CannonInCase(\_)$ denoting which cannon is used in which fire case, and $BelongTo(\_)$ representing which valve belongs to the connection of which cannon.

The operations are described in the ACTION part of Edola by a series of actions: $PowerUp$, $OpenPump$, $SelectCase(i)$, $Confirm$, $HandleControl$, $ClosePump$, $Cancel$ and $PowerDown$. The environmental model is chosen to be generated automatically through the keyword ENV TOTAL in the Edola model. No time constraint is needed in the simplified case.

The 8 requested properties are specified in the TOPROVE part. For example,

$$ClosePumpNotRespond : \text{RESPOND}\ (ClosePump, \quad (6)$$
$$s\_sysState = \text{“cannonOnUse”}, u\_button[\text{“closepump”}]).$$

asserts that the action $ClosePump$ is disabled unless the

user has pressed the button *closepump* as well as the current system state is "cannonOnUse".

Model checking technique can be used to verify the finite instances of a parameterized model. As a result, when choosing TLC model checker as the verification tool in the Edola IDE, a window is popped up to config the parameters. We instantiate the model by 2 fire cases, 2 cannons, and 4 valves. The TLC model checker is then called automatically to check the 8 properties. It generates $66,713$ different states in total and verifies that all the 8 properties hold in 10.0 seconds.

We can also try to prove the 8 properties with the integrated automatic theorem prover Spass. It can prove the properties (if and only if they are inductive invariants) directly on a parameterized Edola model, without the need of instantiation. The result of proving the 8 properties are shown in Table III. The two popular verification methods complement each other and provide the powerful verification capability for Edola.

Table III
THE SPASS PROVING RESULT OF THE 8 PROPERTIES IN EDOLA IDE

Properties	Spass result	Time
1. ClosePumpNotRespond	Proof found.	12.4s
2. SelectCaseNotRepsond	Proof found.	10.6s
3. CaseSelectOnlyOne	Proof found.	1m33s
4. CannonUsedOnlyByOne	Proof found.	39.2s
5. ValveMutex	Proof found.	18m33s
6. OpenPumpAfterPower	Proof found.	1m37s
$OpenPump \Rightarrow$ $s\_buttonLight$["power"]	Completion found.	17.2s
$\Box Inv$	Proof found.	1m26s
$OpenPump \wedge Inv$ $\Rightarrow s\_buttonLight$["power"]	Proof found.	10.3s
7. SelectAfterOpenPump	Proof found.	1m49s
8. ClosePowerAlwaysRespond	Proof found.	16.2s

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel PLC domain-specific modeling language Edola. It provides useful notations to denote the features of PLC like reactivity, scan cycling, real-time and property patterns. As a result, with the Edola language, we can get a better understanding and easier modeling of PLC applications. It is noteworthy that both the two popular automatic verification methods: model checking and automatic theorem proving are supported in Edola modeling. To implement this functionality, we adopt the TLA$^+$ specification language as the intermediate language between Edola and the verification tools, enhancing a large degree of reusability. A prototype IDE for Edola has been implemented, which provides the user-friendly interface for modeling and the seamless integration of two tools TLC and Spass for verification.

As to the future work, we will enrich the Edola language with module compositions, action priorities, etc. to increase its expressiveness. The support of other verification tools like the model checker UPPAAL [12] and the theorem prover CVC3 [13] will also be considered.

ACKNOWLEDGMENT

This research is supported in part by NSFC Programs (No.91018015, No.60811130468) and 973 Program (No.2010CB328003) of China.

REFERENCES

[1] R.W. Lewis. *Programming industrial control systems using IEC 1131-3,volume 50 of Control Engineering Series*. The Institution of Electrical Engineers, Stevenage, United Kingdom, 1998.

[2] F. Bonfatti, P.D. Monari, and U. Sampieri. *IEC 1131-3 Programming Methodology*. CJ International, Fontaine, France, 1999.

[3] R.Wang, X.Song, and M. Gu. Modelling and verification of program logic controllers using timed automata. *IET Software*, 4:127–131, 2007.

[4] Hehua Zhang, Ming Gu, and Xiaoyu Song. Modeling and analysis of stage machinery control systems by timed colored Petri nets. In *Proceedings of the 3rd International Symposium on Industrial Embedded Systems, (SIES 2008)*, pages 103–110, 2008.

[5] G. Canet, S. Couffin, J. J Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In *Proceedings of IEEE International conference on Systems, Man and Cybernetics (SMC'2000)*, pages 2449–2454, 2000.

[6] Hehua Zhang, Stephan Merz, and Ming Gu. Specifying and verifying plc systems with TLA+. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pages 293–294, 2009.

[7] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703, pages 54–66. Springer Verlag, 1999.

[8] The SPASS homepage: http://www.spass-prover.org/index.html[Accessed: 13 Aug. 2011].

[9] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002. See also http://research.microsoft.com/users/lamport/tla/tla.html.

[10] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[11] Hehua Zhang, Ming Gu, and Xiaoyu Song. Specifying time-sensitive systems with TLA+. In *34th Annual IEEE International Computer Software & Applications Conference (COMPSAC 2010)*, pages 425–430, 2010.

[12] The UPPAAL homepage: http://www.uppaal.com/[Accessed: 13 Aug. 2011].

[13] The CVC3 homepage: http://www.cs.nyu.edu/acsys/cvc3/[Accessed: 13 Aug. 2011].

 ISBN: 978-1-61208-165-6