Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 721624, 9 pages
http://dx.doi.org/10.1155/2013/721624

Research Article

Component-Based Formal Modeling of PLC Systems

Rui Wang,[1,2] Yong Guan,[1,2] Luo Liming,[1] Xiaojuan Li,[1,2] and Jie Zhang[3]

[1] *College of Information Engineering, Capital Normal University, Beijing 100048, China*
[2] *National Lab for ISAT, Beijing 100048, China*
[3] *College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China*

Correspondence should be addressed to Rui Wang; rwang04@gmail.com

Received 8 February 2013; Accepted 27 February 2013

Academic Editor: Xiaoyu Song

Functional validation is an important task in complex embedded system. The formal modeling of PLC system for verification is a rough task. Good verification model should be faithful and concise. At one hand, the model must be consistent with the system at the other hand, the model must have suitable scale because of the state explosion problem of verification. This paper proposes a systemic method for the construction of verification model. PLC system architecture and PLC features are modeled as components. This is universal for all PLC applications. We give an automatic translation method for software modeling based on operational semantics. A small example is demonstrated for our approach.

1. Introduction

As embedded control systems are more and more complex, the safety of systems plays a critical role for high dependability. A tiny error may cause financial losses or even cost human lives. Formal methods are an effective way to analyze and assure the reliability of complex systems. Programmable logic controller (PLC), a typical control system, is popular in industry. A PLC controls several processes concurrently. It receives input signals from sensors, processes them, and produces control signals.

Model checking has proved to be a powerful automatic verification technique [1]. It has been successfully applied to hardware design and communication protocol verification. In recent years, this technique has been used to verify a certain type of software and achieved some success. Model checking process has three main steps. First, the system is modeled as a Kripke structure. Then, certain properties are expressed by temporal logic formulas. Model checking algorithm checks if the model satisfies the required properties. If the property is not satisfied, a counterexample is provided. The critical precondition of verification is modeling.

The International Electrotechnical Commission (IEC) published IEC61131 standard [2] for programmable controller. Five PLC program languages defined by IEC are instruction list (IL), ladder diagram (LD), structured test (ST), function block diagram (FBD), and sequential function chart (SFC). Most researches about PLC focus on IL programs. In [3], Canet et al. translate simple IL program into SMV input languages manually. The model is one cycle of the PLC execution, and authors do not consider counters and integer type. Huuck uses abstract interpretation-based static analysis to find running errors in [4]. However, the model is static; only general properties can be checked. Loeis et al. [5] model the control systems cyclic behavior first and then IL programs; they are integrated as one model. SMV is the verification tool. In order to find an automatic translation to formal specification, mealy automaton [6] and XML [7] are used as medial format between programs and verification tool input, but, the program should firstly be rewritten as if-then-else format. Petri net and timed automata are all used to model PLC programs. A PLC program translation tool is given in [8]. It translates IL programs to timed automata which can be checked by *Uppaal* [9]. The data types are restricted to Booleans and do not include function block calls. Heiner and Menzel define a Petri nets semantics of IL in [10], but verification phase is not included. In [11, 12], signal interpreted petri net (SIPN) which extended Petri net with input and output signals is adopted to model PLC system. Such extension is powerful for modeling, but Petri net tool is not strong enough to analyze SIPN; authors still have to use SMV. The methods presented above only consider the software itself. The PLC environment and features of hardware platform are not mentioned.

This paper presents a method of modelling PLC system for verification. The common parts of PLC hardware platform are modeled as BIP (Behavior, Interaction, Priority) [13] components. Function call, timer invoking, and PLC cyclic mode are formalized by BIP synchronization with connectors. This part is same for different PLC applications. We define the operational semantics of PLC instructions. The PLC software is formalized as a transition system according to operational semantics. An example is demonstrated for this modelling procedure. The paper is organized as follows. Section 2 introduces the BIP concepts and related tools. The modeling of PLC architecture and PLC features is shown in Section 3. Section 4 defines the operational semantics of PLC language and the translation-based method of software. In Section 5, we conclude the paper.

2. The BIP Framework

The BIP (Behavior, Interaction, Priority) component framework is a formalism supporting rigorous design for heterogeneous component-based systems [14]. It allows the description of systems as the composition of atomic components characterized by their behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: interactions and priorities. Components are composed by layered application of two operators.

In BIP, atomic components are finite-state automata extended with variables and ports. Variables are used to store local data. Ports are action names and may be associated with variables. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is a step, labeled by a port, from a control location to another. It has associated a guard and an action that are, respectively, a Boolean condition and a computation defined on local variables. In BIP, data and their transformations are written in C. Interactions describe synchronization constrains between ports of the composed components. Interactions are two types: rendezvous (strong symmetric synchronization) and broadcast (weak asymmetric synchronization). Priorities between interactions are used to restrict nondeterminism inherent to parallel systems. BIP separates behavioral and architectural aspects in modelling. Architecture is meaningfully defined as the combination of interactions and priority. Moreover, it presents a discussion about the expressivity of BIP and related component-based frameworks. It shows that the combination of interactions and priorities confers BIP a universal form of expressiveness. Numerous translations are defined from existing models of computation and domain-specific language into BIP.

The BIP framework is concretely implemented by the BIP language and an extensible toolbox [15]. The toolbox provides front-end tools for editing and parsing of BIP programs, as well as for generating an intermediate model, followed by code generation (in C++). Intermediate models can be subject to various model transformations focusing on construction of optimized models for, respectively, sequential [16] and distributed execution [17]. It provides also back-end tools including runtime for analysis (through simulation) and efficient execution on particular platforms. Validation of BIP models can be achieved by using static or runtime validation techniques. The static validation techniques are provided by the D-Finder tool [18]. The runtime validation technique of BIP is based on construction and execution of monitored systems. Monitors are atomic components that observe the system state and react by moving to error state where the safety properties are violated, that is, if an interaction has been executed or an invalid sequence of interactions has been executed.

3. Formalization of PLC Features

3.1. PLC System Architecture. This section proposes the modelling framework for complicated software-hardware mixed system. The execution of software is highly related to the hardware platform and the environment, so we should model hardware platform and the environment. Therefore, PLC system model includes three parts; the software model, hardware platform model, and environment model. PLC hardware platform has the same model and is not related to application software. For the existing PLC software, the model can be obtained by automatic translation. Then the system model can do simulation or verification with the help of BIP tools. This framework is extendible. We can easily add more components.

PLC system architecture shown in Figure 1 is composed of three layers. Software includes all application program organizations. The software is modeled as separate components. Main program can call functions or function blocks. Function block can call nested function block or nested function. *CAL* instruction is modelled as a *CallCon* connectors. The *call* port of calling program component sends signals by broadcast mechanism. It compares the names of all connected components with the names of called functions and decides which one is called. PLC can handle interrupts. The interrupt handler is modelled as a component. Timer is a separate function and is modeled as a component. When timer starts, this component is aroused by *call* port. This layer describes the software structure explicitly.

The second layer is the abstraction model of the hardware platform. This layer simulates the features of PLC, that is, cyclic execution mode and interruption handling.

The bottom layer is environment. In order to make the system closed and available for verification, this layer includes the model of controlled devices. Sensors collect data of environments. This information is written to PLC at the beginning of every execution cycle through *startCyc* port. After the computation of PLC programs, commands are given to actuators through *finishCyc* port. Interrupt events of environment such as communication interrupt, alarm interrupt, and clock interrupt are modelled as components.

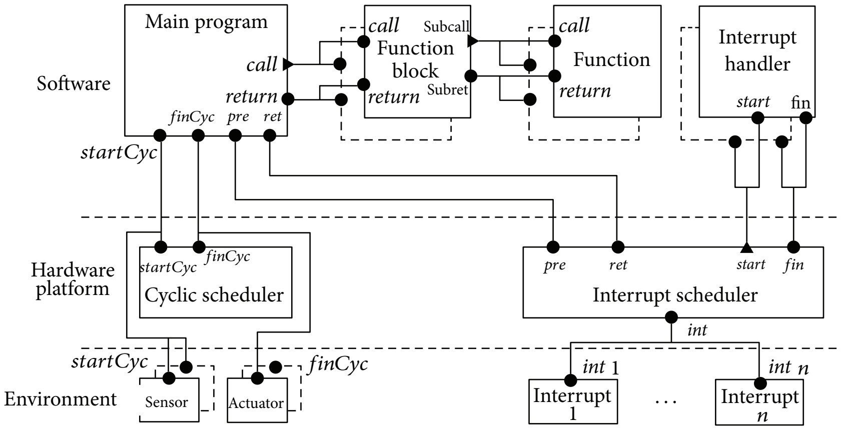


Figure 1: PLC system architecture.

3.2. Formalization of Cyclical Operation Mode. PLC runs in a cyclical way of three stages. At the first stage, it scans signals from the sensors and stores them in the input registers. Then, the instructions in memory are read out and executed. The results are stored in the output registers at the second stage. At last, all the data in the output registers will be output to actuators.

In view of that operation mode, two kinds of models can be extracted. One model at a higher level of extraction ignores the operation details, which is easy to analyze and verify. The other one considers the cyclical operation mode through a scheduling component, which displays the read-in, operation, and read-out of data.

The cyclic scheduler component is shown in Figure 2. It comprises two states. At the beginning, it transmits from the initial state *idle* to the *exe* state, synchronizing with the environment and PLC main program through *startCyc*. The *EXE* state indicates the execution of PLC. After a delay of *CycleTime* which signifies the cycle time, the component moves back to the *idle* state through a synchronization port *finCyc*. That is all for a PLC cycle. Such an explicit model shows the details of the implementation in a cycle. And due to the lower abstraction, we obtain models of a larger scale.

3.3. Formalization of Interrupt Scheduler. Interrupt is a vital feature of PLC. If an interrupt happens, the running program switches to handle it and returns to the original program when finished. PLC admits kinds of interrupts, such as external I/O interrupt, communication interrupt, and time base interrupt. They have different priorities, and the communication interrupt has the top priority. According to the principle of first-come, first-served, a running interrupt is not allowed to interrupt for most PLCs. Until the running one finishes, another interrupt of the highest priority is chosen to execute from interrupt queue. Since the cycle time of PLC is short as tens of milliseconds, in general, interrupts are judged periodically and then get executed.

Figure 3 presents the model of interrupt scheduler model. It answers the *request* signals from hardware and environment. An interrupt ($interrupt_i$) delivers its name to the component that dispatches it. The scheduler component collects all the interrupts in a priority queue and chooses the high priority one to preempt main program by *pre* port. When the component moves to the *Rea* state, it broadcasts scheduling of the interrupt handler, which will be executed by corresponding components in the software model. In that process, the interrupt scheduler can accept new arrivals of interrupts and add them into the queue. When finishing that process, the component transmits to the *PRE* state through a port *fin*. If the queue is empty at that time, it moves back to the initial state and synchronizes with main program by *ret* port. Otherwise, it will continue to handle interrupts.

3.4. Formalization of Function Call. As IEC 61131-3 defines, program organization units (POU) is composed of program, function block (FB), and function, which are the minimum and independent software units in user programs. The PLC softwares organized by POU have good performance on modularity. FB may call functions or other function blocks in a nested way but not recursive. Different from FB, however, function cannot do this owing to no static variables and storage space.

The general pattern of function invoking is presented in this paragraph. The main program calls functions through a broadcast port *call* with parameters of FBid which is the name of FB component to communicate with the function arguments which will be valued. As shown in Figure 4, the called component runs after receiving *call* signal. When it comes to the RET instruction at the end, a stop signal will be sent out through *ret* port to the program that makes the call.

3.5. Formalization of Timer. Real time is a significant feature for embedded system. PLC has strict time constraints as well, which is implemented by an internal timer. A special signal *tick* is introduced to model clock. That definition is similar to the clock variables in timed automata. Here, *tick* works with a fixed frequency. All the components that concern about time have such a strong synchronization signal *tick*.

Three types of timer are included in PLC: TON, TONRs and TOF. These three timers have equivalent function, although suitable for different scenarios. The most commonly

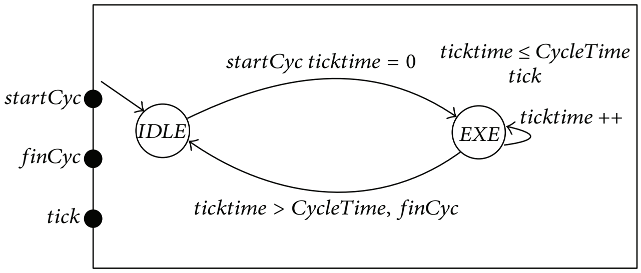


Figure 2: BIP model of cyclical scheduler.

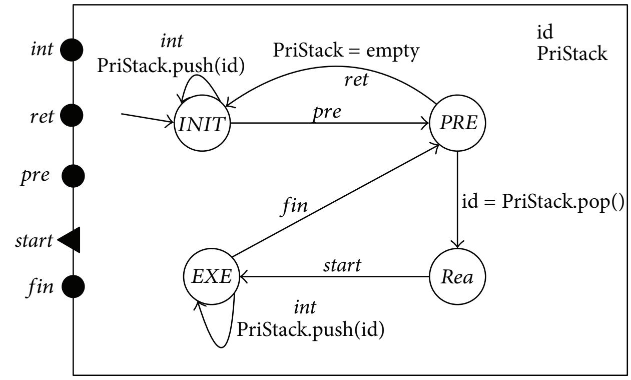


Figure 3: BIP model of interrupt scheduler.

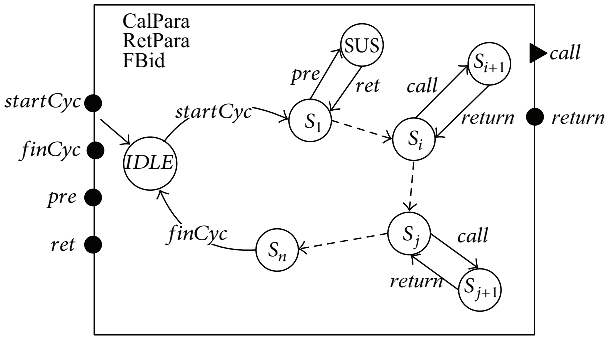


Figure 4: BIP model of function call.

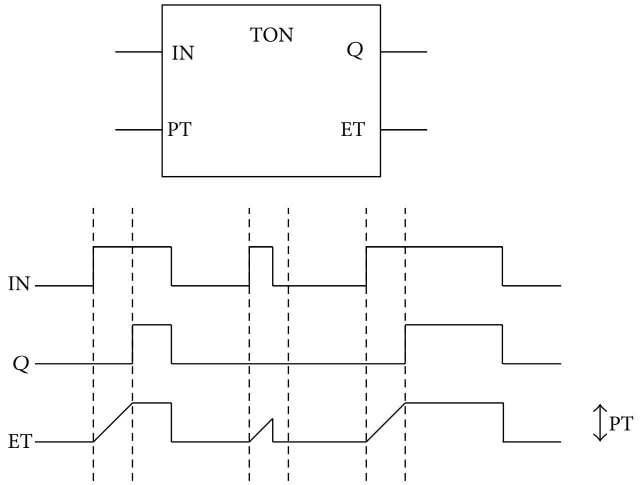


Figure 5: TON timer.

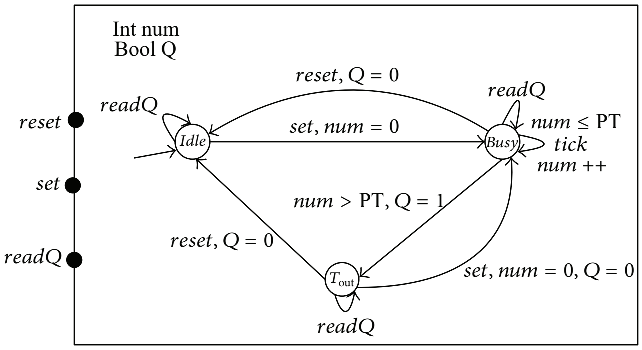


Figure 6: BIP component of TON timer.

used TON will be discussed in this paper. In IEC 61131-3, the TON and sequence chart are illustrated in Figure 5. The input port IN is enabled, and the input port for integers PT provides the preset value for the timer. The output Q denotes whether the timer reaches the preset value. Current time is measured by an output ET. When IN becomes true, the timer gets started. ET will increase as time elapses. When it increases to PT, Q keeps true until IN turns to false.

Mader and Wupper [19] have given the equivalent PLC function block and the timed automata model for TON timer instruction. They used the signal synchronization and shared data to implement the timer model. BIP language is more safe because it does not support shared variables. The input and output of timer is modeled as ports. Preset value is the parameter of timer component.

PLC-BIP model of timer is shown in Figure 6. Timer works together with PLC programs. The enabled input variable is modeled by assigning port *set* to 1 and *reset* port to 0. Event *read* Q happening at any state can read the value of Q. The component is at *Idle* state initially; when receiving *set* signal, it transmits to *Busy* state and assigns *num* to 0. State *Busy* indicates that the timer has started. When synchronized with *tick*, the value of *num* increases to 1. If the *num* is larger than PT, the component transmits to *Timeout* state and set Q to 1.

4. Translation-Based Modeling of Software

For the existing system, the main program and functional block in Figure 1 can be achieved by automatic translation. The main program and functional block are translated to automatic components. We define the connectors for function calls. Software models are composed by these automatic components and connectors. The system model obtained by this method has kept the topology structure of software. This section introduces the IL instructions of PLC, defines the operational semantics of these instructions, and proposes the translation method and rules.

4.1. IL Instructions. In order to make this method more common, we choose IL language defined in IEC 61131-3 as the source code. IEC 61131-3 defines the modifier, function, and function block. Compared with other PLC languages, IL is more concise and assembly-like text language. IL language supports bool, integer, and float. The (current result) cr register stores current computing result. Some instructions are related to the value of cr.

Timer is implemented by hardware. IEC 61131-3 defines the timer as a system function call. When starting a timer, the program uses *CAL* instruction. Except for timer, other instructions are all real time independent. Our method models PLC POU as an atomic component. The calling of interrupt handler is similar to function call.

(i) Bit logic instructions: AND, OR, XOR, and NOT.

(ii) Set and reset instructions: S, R.

(iii) Data load and transfer instructions: LD, ST.

(iv) Logic control instructions: JMP, CAL, and RET.

(v) Integer math instructions: ADD, SUB, MUL, DIV, and MOD.

(vi) Comparison instructions: GT, GE, EQ, NE, LE, and LT.

IL instruction can have one operand or none. The operands of instructions can be variable, constant, label, or address. Table 1 shows the meaning of common IL instructions. There are three kinds of variables: I is the input variable, Q is the output variable, and M is the local variable.

4.2. The Semantics of IL Instructions. The PLC programming organization unit $\mathscr{P}$ has three types; program (*Prog*), function (*Fun*), and function block (FB). Program configuration is the program execution environment including all data of the program.

Definition 1. The configuration of programming organization unit $\mathscr{P}$ is $C_P = \langle \mathrm{ID}, \mathrm{PC}, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle$:

(i) ID is the name of current execution program,

(ii) PC is the program counter,

(iii) V is the set of variables, including cr, cr $\in V$,

(iv) P_{IN} is the variables of input port of program $\mathscr{P}$. If $\mathscr{P}$ has the type of *Prog*, this port is synchronous with the cyclic component with *startCyc* port. If $\mathscr{P}$ is FB type, this port is synchronous with *call* port,

(v) P_{OUT} is the variables of the output port of $\mathscr{P}$. If $\mathscr{P}$ has the type of *Prog* this port is synchronous with the cyclic component with port *finishCyc*. If $\mathscr{P}$ is FB type, this port is synchronous with *ret* port.

IL program $\mathscr{P}$ is a sequence of instructions $l_1, l_2, \ldots, l_m$, where $m \in \mathbb{N}$ is the number of $\mathscr{P}$. For any instruction l_i, the operational semantics $\mathscr{S}[\![l_i]\!]$ is a transition system. The program configuration is the state, and the execution of an IL instruction causes a state transition from one configuration to another configuration. We define the BIP component model of program as follows.

Definition 2. Transition system is a triple $\Delta = \langle C_P, T, C_P^0 \rangle$, where

(i) C_P is PLC program configuration,

(ii) $T \subseteq C_P \times C_P$ is the set of transition relations,

(iii) $C_P^0 \in C_P$ is the initial state.

For the common denotation of all instructions, we add an IO instruction at the beginning with PC assigning 0. This instruction is used for synchronization with *startCyc* port and *call* port. It does not have data operation. The initial configuration is $\langle \mathrm{ID}, 0, V^{\mathrm{init}}, P_{\mathrm{IN}}^{\mathrm{init}}, P_{\mathrm{OUT}}^{\mathrm{init}} \rangle$.

(1) The operational semantics of input instruction $\mathscr{P}(0) = \mathrm{IO}$ is defined as follows. If $\mathscr{P}$ is *Prog* type, the data of port is transmitted. If the type is FB, the real parameter is passed by ports. "$\mapsto$" denotes the change of variables. $\overrightarrow{I}$ means the data vector of port. $\overrightarrow{startCyc(\mathscr{P})}$ means combining data vector with input port of program $\mathscr{P}$, if the type of $\mathscr{P}$ is *Prog*. Therefore,

$$\mathscr{S}[\![io]\!] = \frac{\mathrm{PC}' = 1,\ P'_{\mathrm{IN}} = P_{\mathrm{IN}}\left[\overrightarrow{I} \longmapsto \overrightarrow{startCyc\,(\mathscr{P})}\right]}{\langle \mathrm{ID}, 0, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle \longrightarrow \langle \mathrm{ID}, \mathrm{PC}', V, P'_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle}. \tag{1}$$

If $\mathscr{P}$'s type is FB, then

$$S[\![io]\!] = \frac{\mathrm{PC}' = 1,\ P'_{\mathrm{IN}} = P_{\mathrm{IN}}\left[\overrightarrow{I} \longmapsto \overrightarrow{call\,(\mathscr{P})}\right]}{\langle \mathrm{ID}, 0, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle \longrightarrow \langle \mathrm{ID}, \mathrm{PC}', V, P'_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle}. \tag{2}$$

(2) If $\mathscr{P}(\mathrm{PC}) = \mathrm{AND}$ op, the operational semantics is

$$\mathscr{S}[\![\mathrm{AND}]\!] = \frac{\mathrm{PC}' = \mathrm{PC} + 1,\ V' = V\left[\mathrm{cr} \longmapsto \mathrm{cr} \land \mathrm{op}\right]}{\langle \mathrm{ID}, \mathrm{PC}, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle \longrightarrow \langle \mathrm{ID}, \mathrm{PC}', V', P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle}. \tag{3}$$

This instruction only changes the value of program counter and cr. Other logical instructions such as OR, XOR, and NOT have the similar operational semantics. The type of op is `BOOL`.

(3) If $\mathscr{P}(\mathrm{PC}) = S$ op, the operational semantics is

$$\mathscr{S}[\![S]\!] = \frac{\mathrm{PC}' = \mathrm{PC} + 1,\ V' = V\left[\mathrm{if}\,(\mathrm{cr} = 1)\ \mathrm{op} \longmapsto 1, \mathrm{else}\ \mathrm{op} \longmapsto 0\right]}{\langle \mathrm{ID}, \mathrm{PC}, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle \longrightarrow \langle \mathrm{ID}, \mathrm{PC}', V', P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle}. \tag{4}$$

The value of cr is the execution condition. If cr is 1 the operand is set to 1; otherwise, operand is set to 0.

(4) If $\mathscr{P}(\mathrm{PC}) = \mathrm{LD}$ op, assign the value of op to register cr. Therefore,

$$\mathscr{S}[\![\mathrm{LD}]\!] = \frac{\mathrm{PC}' = \mathrm{PC} + 1,\ V' = V\left[\mathrm{cr} \longmapsto \mathrm{op}\right]}{\langle \mathrm{ID}, \mathrm{PC}, V, P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle \longrightarrow \langle \mathrm{ID}, \mathrm{PC}', V', P_{\mathrm{IN}}, P_{\mathrm{OUT}} \rangle}. \tag{5}$$

Table 1: The meaning of IL instructions.

Instruction	Modifier	Type	Description
AND	N,(	Variable, constant	Logical AND
OR	N,(	Variable, constant	Logical OR
XOR	N,(	Variable, constant	Logical XOR
NOT		None	Logical NOT
S		Variable	Set
R		Variable	Reset
LD	N	Variable, constant	Assign the value of operand to cr
ST	N	Variable	Assign the value of cr to operand
JMP	C,N	Label	Jump to label instruction
CAL	C,N	Function name	Function call
RET	C,N	None	Function return
ADD	(	Variable, constant	Add operation
SUB	(	Variable, constant	Subtraction operation
MUL	(	Variable, constant	Multiply operation
DIV	(	Variable, constant	Division operation
MOD	(	Variable, constant	Mode operation
GT	(	Variable, constant	Compare the result is BOOL

(5) If $\mathscr{P}(\text{PC}) = \text{ADD op}$, this math instruction assigns the value of op with cr and saves it to cr. The semantics of other math instructions are similar. Therefore,

$$\mathcal{S}[\![\text{ADD}]\!] = \frac{\text{PC}' = \text{PC}+1,\ V' = V\left[\text{cr} \longmapsto \text{cr}+\text{op}\right]}{\langle \text{ID}, \text{PC}, V, P_{\text{IN}}, P_{\text{OUT}}\rangle \longrightarrow \langle \text{ID}, \text{PC}', V', P_{\text{IN}}, P_{\text{OUT}}\rangle}. \tag{6}$$

(6) If $\mathscr{P}(\text{PC}) = \text{GT op}$, compare instruction compares the operand with cr, the BOOL result is saved in register cr. Therefore,

$$\mathcal{S}[\![\text{GT}]\!] = \frac{\text{PC}' = \text{PC}+1,\ V' = V\left[\text{if } (\text{cr} > \text{op})\ \text{cr} \longmapsto 1, \text{else } \text{cr} \longmapsto 0\right]}{\langle \text{ID}, \text{PC}, V, P_{\text{IN}}, P_{\text{OUT}}\rangle \longrightarrow \langle \text{ID}, \text{PC}', V', P_{\text{IN}}, P_{\text{OUT}}\rangle}. \tag{7}$$

(7) If $\mathscr{P}(\text{PC}) = \textit{JMPC label}$ and cr is 1, then jump to instructions with the name of *label*; otherwise, execute the next instruction. Therefore,

$$\mathcal{S}[\![\textit{JMPC}]\!] = \frac{\text{if } (\text{cr} = 1)\ \text{PC}' = \text{label},\ \text{else } \text{PC}' = \text{PC}+1}{\langle \text{ID}, \text{PC}, V, P_{\text{IN}}, P_{\text{OUT}}\rangle \longrightarrow \langle \text{ID}, \text{PC}', V, P_{\text{IN}}, P_{\text{OUT}}\rangle} \tag{8}$$

(8) If $\mathscr{P}(\text{PC}) = \text{CAL op}$, here op is the name of called POU; operand is passed by the first instruction IO. Therefore,

$$\mathcal{S}[\![\text{PC}]\!] = \frac{\text{ID}' = \text{op},\ \text{PC}' = 0}{\langle \text{ID}, \text{PC}, V, P_{\text{IN}}, P_{\text{OUT}}\rangle \longrightarrow \langle \text{ID}', \text{PC}', V, P_{\text{IN}}, P_{\text{OUT}}\rangle}. \tag{9}$$

(9) If $\mathscr{P}(\text{PC}) = \text{RET}$, return instruction gives the result to calling program through connectors and ports. *pre* (PC) is the value of calling program. *pre* (ID) is the name of calling program. Therefore,

$$\begin{aligned} \mathcal{S}[\![\text{RET}]\!] &= \Big(\text{PC}' = \textit{pre}\,(\text{PC}) + 1, \text{ID}' = \textit{pre}\,(\text{ID}),\\ &\qquad P'_{\text{OUT}} = P_{\text{OUT}}\left[\overrightarrow{O} \longmapsto \overrightarrow{\textit{finCyc}\,(\mathscr{P})}\right]\Big)\\ &\quad \times \left(\langle \text{ID}, \text{PC}, V, P_{\text{IN}}, P_{\text{OUT}}\rangle \longrightarrow \left\langle \text{ID}', \text{PC}', V, P_{\text{IN}}, P'_{\text{OUT}}\right\rangle\right)^{-1}. \end{aligned} \tag{10}$$

4.3. Automatic Translation Rules. The instruction semantics explains the execution effect of the configuration. We can extract the translation rule in line with instruction semantics.

Assuming that program $\mathscr{P}$ is composed of n instructions then $\mathscr{P} = \{\text{IO}, l_1, \ldots, l_n\}$. The initial state of the translation system is $\langle \text{ID}, 0, V^{\text{init}}, P_{\text{IN}}^{\text{init}}, P_{\text{OUT}}^{\text{init}}\rangle$. The transition for instruction l_i is $C_p^i \xrightarrow{exe(l_i)} C_P^{i+1}$.

PLC program control instruction will change the structure of the transition system. We conclude these instructions into four kinds as shown below. *stm* stands for one instruction and *code* is a segment of instructions.

(1) Basic instructions

$$\textit{Code} = (\textit{stm}_i), \tag{11}$$

The state machine for this kind of instruction is shown in Figure 7.

(2) Sequence instructions

$$\textit{Code} = \begin{pmatrix} \textit{stm}_i \\ \textit{stm}_{i+1} \end{pmatrix}, \tag{12}$$

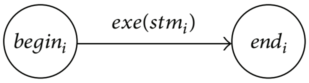


Figure 7: Basic instruction translation rule.

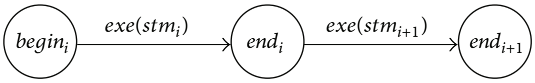


Figure 8: Sequence instruction translation rule.

Sequence instructions are two instructions executed one by one. Figure 8 combines the finishing state of stm_i with the beginning state of stm_{i+1}.

(3) Branch instruction

$$Code = \begin{pmatrix} & JMP\,(C) & label \\ & code_1 & \\ label & code_2 & \end{pmatrix}. \tag{13}$$

Jump instruction is used for branching control. *JMP* instruction is for uncondition jump. The program will directly jump to $code_2$. When the value of cr is 1, *JMPC* instruction will jump; otherwise, it executes the next instruction (see Figure 11). Figure 9 models jump instructions.

(4) Function call instruction

$$Code = \begin{pmatrix} CAL & \text{FB name} \\ code_1 & \end{pmatrix}, \tag{14}$$

In BIP model, *CAL* instruction is synchronous with called component through *call* port. When the called function finished execution, it returns to the main program with values through *ret* port.

While translating according to the rules strictly, the state space is large. The transition for sequence instruction only changes the value of local variable and dose not communicate with other components through ports. For example, transitions $C_p^i \xrightarrow{r(l_i)} C_p^{i+1} \xrightarrow{r(l_{i+1})} C_p^{i+2} \xrightarrow{r(l_{i+2})} C_p^{i+3}$ are all internal transitions. BIP is a high-level modelling language and expressiveness. Transitions in BIP component always have communication signals. So when the program segments only have sequence instructions, we can compress these steps into one step, that is, $C_p^i \xrightarrow{r(l_i);r(l_{i+1});r(l_{i+2})} C_p^{i+3}$. One transition has three assigned operations.

In conclusion, the steps of translation-based modelling method are as follows.

(1) Translate the program organization units into atomic components.

(2) Define the type of connectors based on the communication ports.

(3) Instantiate atomic components and connectors.

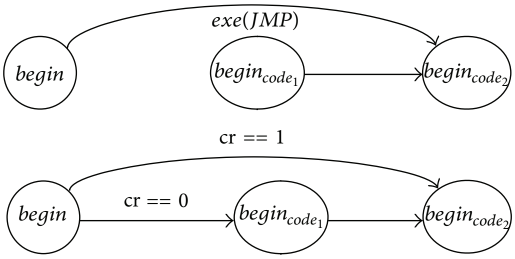


Figure 9: Branch instruction translation rule.

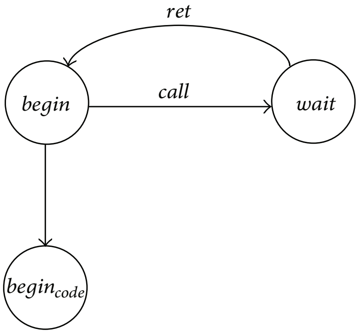


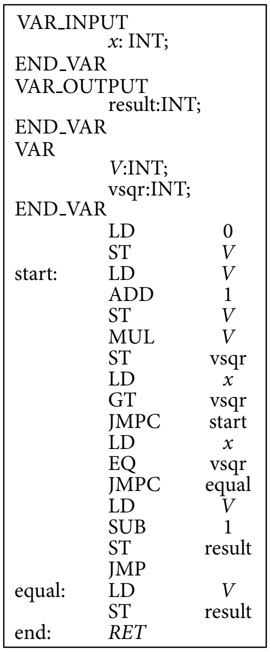
Figure 10: Function call instruction translation rule.

(4) Compose software model, platform model, and environment model into a compound component (see Figure 10).

Here is an example demonstrating the translation-based modelling method. Figure 12 is the IL program for computing the square root. Figure 2 is the corresponding formal models. This component has two ports: calling port *call* and returning port *call*. Port *call* binds the input data x, and port *ret* binds the square root of x. The segments without jump instruction and call instruction can be compressed into one transition. This method reduces the scale of model.

5. Conclusion

Computer-aided verification is an important task in complex embedded system. The formal modelling of PLC system for verification is a rough task. At one hand, the model must be faithful with the system; at the other hand, the model must have suitable scale because of the state explosion problem of verification. This paper has proposed a systemic method for the construction of verification model. PLC system architecture and PLC features have been modelled as components. This is universal for all PLC applications. The operational semantics of PLC instructions have been formally defined. We have given an automatic translation method for software modelling based on operational semantics. The automatic translation method ensures that the model is consistent with the source code. A small example has been demonstrated for our approach.



```
VAR_INPUT
            x: INT;
END_VAR
VAR_OUTPUT
            result:INT;
END_VAR
VAR
            V:INT;
            vsqr:INT;
END_VAR
            LD          0
            ST          V
start:      LD          V
            ADD         1
            ST          V
            MUL         V
            ST          vsqr
            LD          x
            GT          vsqr
            JMPC        start
            LD          x
            EQ          vsqr
            JMPC        equal
            LD          V
            SUB         1
            ST          result
            JMP
equal:      LD          V
            ST          result
end:        RET
```

Figure 11: IL program.

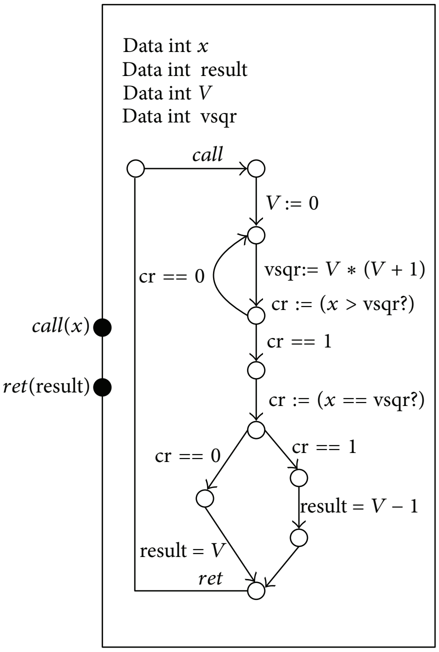


Figure 12: Program model.

Acknowledgments

This work is supported by the International S&T Cooperation Program of China (2011DFG13000), Mechanism and Verification of High-speed Embedded Communication Systems in Rugged Environment (2010DFB10930), and the Beijing Natural Science Foundation and S&R Key Program of BMEC (4122017, KZ201210028036).

References

[1] E. M. Clarke and O. Grumberg, *Model Checking*, The MIT Press, Cambridge, Mass, USA, 1999.

[2] International Electrotechnical Commisson, Techincal Committee No 65, Programmable Controller-Programming Languages, IEC 61131-3, 2nd edition, comminttee draft, 1998.

[3] G. Canet, S. Couffin, J. J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pp. 2449–2454, October 2000.

[4] R. Huuck, "Semantics and analysis of instruction list programs," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 3–18, 2005.

[5] K. Loeis, M. B. Younis, and G. Frey, "Application of symbolic and bounded model checking to the verification of logic control systems," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '05)*, vol. 1, pp. 247–250, Catania, Italy, September 2005.

[6] M. B. Younis and G. Frey, "Formalization of PLC programs to sustain reliability," in *Proceedings of the IEEE Conference on Robotics, Automation and Mechatronics (RAM '04)*, pp. 613–618, Singapore, December 2004.

[7] M. B. Younis and G. Frey, "Visualization of PLC programs using XML," in *Proceedings of the American Control Conference (AAC '04)*, pp. 3082–3087, Boston, Mass, USA, July 2004.

[8] H. X. Willems, "Compact Timed Automata for PLC Program," Tech. Rep., University of Nijmegen, 1999.

[9] http://www.uppaal.com/.

[10] M. Heiner and T. Menzel, "A petri net semantics for the PLC language instruction list," in *Proceedings of the IEE Workshop on Discrete Event Systems*, pp. 161–166, 1998.

[11] T. Mertke and G. Frey, "Formal verification of PLC-programs generated from Signal Interpreted Petri Nets," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pp. 2700–2705, usa, October 2001.

[12] X. Weng and L. Litz, "Verification of logic control design using SIPN and model checking—methods and case study," in *Proceedings of the Americal Control Conference*, pp. 4072–4076, June 2000.

[13] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM '06)*, IEEE Computer Society, 2006.

[14] A. Basu, B. Bensalem, M. Bozga et al., "Rigorous component-based system design using the BIP framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[15] "The BIP Toolset," http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.

[16] M. Bozga, M. Jaber, and J. Sifakis, "Source-to-source architecture transformation for performance optimization in BIP," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 708–718, 2010.

[17] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT '10)*, pp. 209–218, October 2010.

[18] S. Bensalem, M. Bozga, T. H. Nguyen, and J. Sifakis, "D-finder: a tool for compositional deadlock detection and verification,"

in *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, pp. 614–619, 2009.

[19] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 106–113, 1999.