A User-oriented Development Method in Multiprocessor Embedded PLCs for Complex Logic and Motion Control Mixed Scenarios

***Abstract*—Programmable logic controllers (PLCs) are a foundation of automation, but applications become complex in logic and motion control mixed scenarios, while a PC-based PLC incurs a high monetary cost and comprises a complex system that cannot meet the customized requirements of large equipment. The development of PLCs has encountered bottlenecks. Hence, in this paper, we theoretically present a user-oriented development method containing a comprehensive optimization method. In practice, we implement this concept by proposing a customized multiprocessor embedded PLC ($ePLC$) to enhance the performance, a multi-language supported uniform development platform to improve the adaptability of developers, and an optimized system structure (reasonable memory allocation, user-oriented thread structure, proposed LPM data interaction, modular software design, and finite state machines) to reduce the development complexity. Ultimately, we adopt the proposed method to implement a distributed control system on a 200-T injection molding machine. Through comparison with TECHMATION and KEBA systems, the startup time of the implemented system was increased by more than 5 times, while the key performance is almost identical. In addition, the implemented system adopts the customized multiprocessor embeded PLC and detached human machine interface (HMI).**

***Index Terms*—Multiprocessor, motion control, injection molding machine, embedded PLC, user-oriented**

I. Introduction

Some concepts, such as smart factory and intelligent manufacturing [?], [1], and some technologies, such as the Internet of Things, 5G, and augmented reality [?], [?], [?], are paving the road of the fourth industrial revolution. Normally, a typical plant is full of large equipment; for instance, cranes, computerized numerical control (CNC) machining centers, injection molding machines (IMMs), air pumps, chillers, automatic-guided vehicles, and various types of robots, and most of them are controlled by programmable logic controllers (PLCs), which have become a primary control system. Numerous researchers are focusing on PLC technologies that significantly extend the application fields of PLCs. [2]–[4] guaranteed PLC reliability by verifying their programming, [5]–[7] improved the performance of PLCs using advanced algorithms, [?] alleviated the development complexity of PLCs using special software structure, and [8], [9] proposed methods of updating $PLCs$ programs dynamically. However, with the rapidly growing demand for, and the trends of, applications to be user-oriented and mix complex logic and motion control [10], [11], PLCs still encounter bottlenecks, especially on large equipment (CNC machining centers, $IMMs$, etc.).

A. Motivations

Currently, the hardware architecture of PLCs can take two directions: embedded PLCs ($ePLC$s) and PC-based PLCs. PC-based PLCs are increasingly used in applications of complex logic and motion control mixed scenarios on account of its high performance and numerous user-oriented tools [11]. Considering the IMM industry, Table I lists the composition of IMMs, including parts (described as modules for programming), digital input\output (DI\Os), and analog input\output (AI\Os). Normally, the simplest IMM system consists of 10 modules, 20 DIs, 30 DOs, three AIS, and seven AOs. Complex relations among them and the high-performance requirements of algorithms tremendously increase the difficulty of programming. Hence, as listed in Table II, a comparison of KEBA, BECKHOFF, GAFRAN, and TECKMATION systems, which are the main brands of IMM controllers, illustrates that almost all of them use PC-based PLCs. In the PC-based PLC, the runtime executes in some cumbersome systems (e.g., VxWorks, Windows CE, RTLinux). Additionally considering the above-mentioned complexity of an IMM system and the software architecture of PC-based PLCs, it has become a huge project to build their own IMM system for the manufacturers; hence, the hardware and software of the controller are always fixed and the HMI is also integrated with the PLC. This situation leads to little independence among IMM manufacturers and the development of PLCs in these scenarios has encountered bottlenecks.

$ePLC$s have a wide area of application in automation due to their easy programming and high reliability, which induces a large paradigm shift of $PLCs$ to the embedded processor market. Some Platforms (e.g., OpenPLC [?]) are transferred to some market-friendly embedded hardware (e.g., Raspberry-Pi) to improve acceptability. Some light-weight operating systems (e.g., uClinux [?]) are adopted in embedded processors to simplify the development of an embedded system. Some methods, such as model- and component-based software [12], [13], have also been researched to reduce the complexity of PLC programs. However, some disadvantages still limit its further development [11], especially when coping with complex logic and motion control mixed applications. A more comprehensively improved development method combining these above-mentioned advances still should be proposed. Hence, how to integrate easy programming and the high reliability of $ePLCs$ with high performance becomes crucial for logic and motion control mixed scenarios.

B. Related Works

To address the problem of complex logic and motion control mixed scenarios, various researchers have presented methods of integrating motion control algorithms (e.g., linear interpolation, position control, arc interpolation, etc.) into PLCs [14], [15], and thus logic control and motion control become inseparable. There are two ways to realize this integration: 1) an individual motion control module integrated with the PLC [16], and 2) PLCs featuring motion control functions [?], [?], [17]. Regarding an individual motion control module, different kinds of modules and PLCs coded in different languages and developed on their own platforms tremendously increase the complexity of implementing applications. [16], [18], [19] all describe these modules. The second way simplifies the development method; nevertheless, it is difficult to guarantee high-reliability logic control and high-accuracy motion control simultaneously, since they run in the same thread.

We propose the concept of a multiprocessor $ePLC$ (multiple processor chips and multiple cores in one chip are all called multiprocessors in this paper). In this $ePLC$, extra processors (e.g., DSP, FPGA, etc.) are introduced to enhance performance. Various technologies contribute to the improvement of multiple processors. [20], [21] proposed data-interaction methods among multiple processors, [22] presented a method to balance the computing ability of the processors, [23] proposed a thread-scheduling method in multiprocessors. All of these works are not implemented in the $ePLC$, but have inspired us to build the architecture of a multiprocessor $ePLC$. Moreover, some research [24], [25] introduced additional high-performance processors into $ePLC$s, although no improvement in the development method was proposed for complex logic and motion control mixed applications.

In terms of the development methods of the individual module, which is very convoluted, users should take considerable time in selecting the platform and additional time in learning the particular software and its supported language. For instance, PMAC uses C++ [16], [18], MC421/221 of the OMRON CS1 series is supported by G-Code [26], the FP series PLC of Panasonic adopts special instructions embedded in a ladder diagram (LD). Therefore, the uniform development methodology in PLC platforms has attracted our interest. Since the 1990s, IEC-61131 has been focused on the standardization of PLCs [27]. In 2005, the PLCopen organization released a related standard [28] that standardizes the motion control in PLCs and then publications, such as [29], suggested interactions using it, and companies, such as 3S [30], provided some tools. However, regarding such complex applications, programmers occasionally prefer to use more popular or object-oriented languages (e.g., C, C++, etc.) [31]–[33], rather than the languages specified in IEC-61131-3.

C. Our Contributions

Considering the popular user-oriented concept [34], [35], in this work we theoretically develop a concept of a user-oriented development method. To the best of our knowledge, a user-oriented development method should improve every aspect of the PLC system (development method, program, processor, RAM, and threading) and contain a comprehensive optimization approach **to address specific problems from the user's point of view**. Hence, we pose a flexible solution to enhance performance by adding sufficient processors, a multi-language-supported graphical component to improve the adaptability of developers, an optimized system structure (reasonable memory allocation, user-oriented thread structure, LPM data interaction, modular software design, and finite state machines) to reduce the development complexity. Ultimately, we adopt the proposed method to implement a distributed IMM system that is considered a kind of complex logic and motion control mixed application.

TABLE I
MODULES, DI/DO, AI/AO OF IMM

No.	Module	DI	DO	AI	AO
1	Mold	Safety valve	Mold close	Mold position	System pressure
2	Injection	Heating detection	Mold open	Injection position	System flow
3	Core	Servo alarm	Inject	Nozzle position	Back pressure
4	Nozzle	Motor overload	Charging	Temperature 1	
5	Heating	Emergency button	Grean light	Temperature 2	
6	Ejector	Injection shield	Red light	Temperature 3	
7	AirValve	Detection switch	Yellow light	Temperature 4	
...	....	...	...	...	...
50		Screw speed			

TABLE II
SYSTEM COMPARISON OF PC-BASED PLC IN IMM

Brand	CPU	System	Language	Distributed	HMI
TECHMATION	DSP	-	Assembly	No	Irreplaceable
KEBA	PC-based	VxWorks	IEC61131-3	Yes	Irreplaceable
BECKHOFF	PC-Based	WindowsCE	IEC61131-3	Yes	Irreplaceable
GEFRAN	PC-based	VxWorks	IEC61131-3	Yes	Irreplaceable

This rest of this paper is organized as follows. In Section II, we introduce the system architecture, multi-language supported graphical component, memory allocation, user-oriented multi-threading, and modular design. In Section III, we present the compilation of graphical components, the LPM data interaction mechanism, the execution of multi-threading, and finite state machines. Finally, in section IV, we implement the IMM distributed system with the proposed method and compare it with the TECHMATION and KEBA systems from aspects of system condition, system structure, and key performance.

II. SYSTEM ARCHITECTURE

A. Hardware Structure of $ePLC$

Figure 1 shows one type of hardware structure of a multiprocessor $ePLC$ that contains a master processor and two slave processors. The master processor is responsible for logic control, communication, etc. The slave processor is designed for complex algorithms that can be customized on demand (the

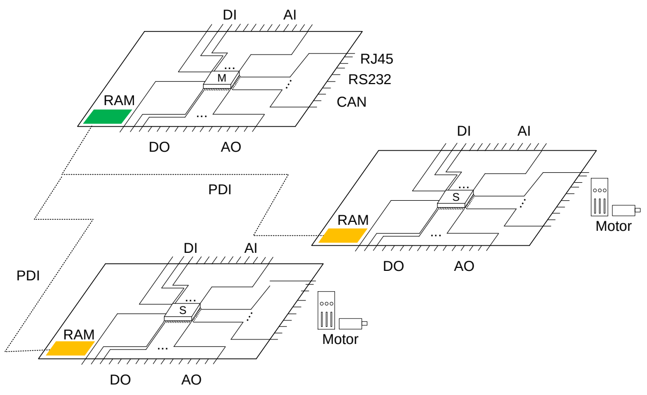


Fig. 1. One type of hardware structure of multiprocessor $ePLC$ containing a master processor and two slave processors.

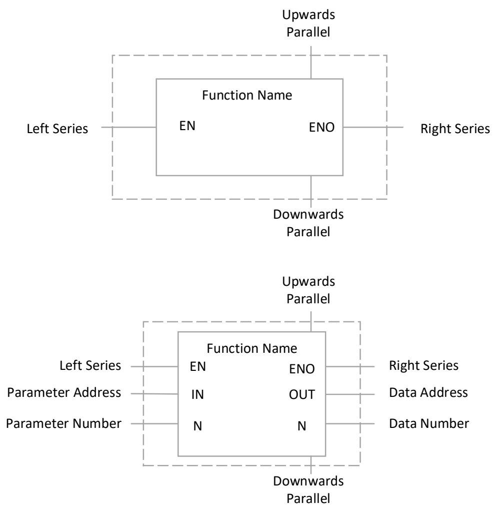


Fig. 2. Two typical designs: single component and component with input and output.

number of processors, DI\Os, AI\Os, and controlled servo motors).

B. Multi-language supported graphical component

In order to develop the logic program and algorithm program on a uniform platform, we package the algorithm into graphical component that is multi-language-supporting. In technical implementation, the formal description is crucial for compiling the graphical component of the instruction list. Hence, the component is described as a 6-tuple:$\{Name, ID, PI, RI, PT, SF\}$. Here, **Name** is the name of component used to describe the function; **ID** is the unique identifier of the component in the graphical program; **PI** is a collection of service interfaces, including output data interfaces, right serial connection, downwards parallel connection, and some auxiliary interfaces; **RI** is a collection of requirement interfaces, including input data interfaces, left serial connection, upwards parallel connection, and some auxiliary interfaces; **PT** is an attribute collection of the component, including position, size, comment, etc.; and **SF** is the

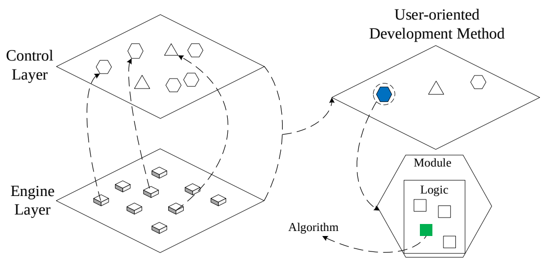


Fig. 3. From the user's point of view, after inducing the multi-language component, the algorithm and logic program can be developed on a uniform PLC platform.

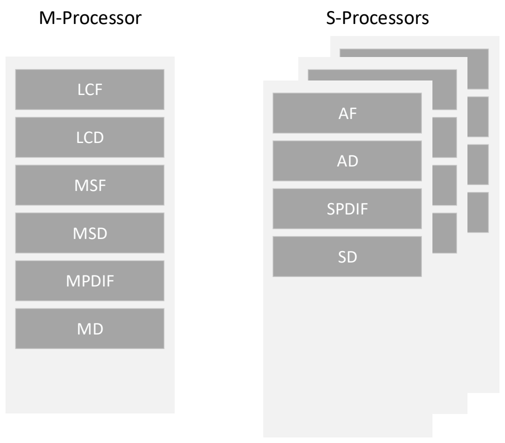


Fig. 4. Memory allocation in master and slave processors.

function description explained by specific text, formula ,or frame template. The basic graphical components are divided into contact components, functional block components, coil components, cross-line vertical components, change lines, comments, etc. The multi-language component, in particular, includes the development language, the supported compiler, and the executing processor.

Figure 2 illustrates two component designs: a single component and a component with input and output. The single component has function name, left series, right series, upwards parallel, and downwards parallel. The component with input and output contains function name, left series, right series, upwards parallel, downwards parallel, parameter address, parameter number, data address, and data number.

As shown in Fig. 3, from the user's point of view, after inducing the multi-language component, the algorithm and logic program could be developed in a uniform PLC platform. Multi-language components are supported by multiple languages, such as IL instructions, ST language, C language, C++ language, etc. Algorithms contained in components are mainly motion control algorithms.

C. Memory Allocation

The dedicated storage area of a PLC in memory is made up of a bit data area (M area) and byte data area (D area). Meanwhile, we regard M area and D area as set M of bit and set D of byte. Furthermore, in the rest of this paper, if $\exists$ set T, we describe its subscript lower-case letter t_i as an element of T, and the subscript i is used to distinguish the elements. Henceforth, two definitions are illustrated below.

Definition 1 If $T \subseteq M$, the value saved in $t_i \in \{0, 1\}$ and each element t_i has four operators: $\mathcal{S}_0(t_i)$ denotes that t_i is set to 0, $\mathcal{S}_1(t_i)$ denotes that t_i is set to 1, $\mathcal{J}_0(t_i)$ represents that the value of t_i is judged as 0, and $\mathcal{J}_1(t_i)$ represents that the value of t_i is judged as 1. We then define the set T as having $\mathcal{B}$ attribute.

Definition 2 If $T \subseteq D$ and $\forall t_i \in T$ has 4 bytes. We define the set T as having $\mathcal{D}$ attribute.

Figure 4 shows the memory allocation of master and slave processors. All slave processors have the same storage structure.

LCF (Logic Control Flag Area): the flags are used to start the modules. It has $\mathcal{B}$ attribute.

LCD (Logic Control Data Area): these data will be used to deliver to the algorithm. It has $\mathcal{D}$ attribute.

AF (Algorithm Flag Area): includes the algorithm flag of execution (AFE) and algorithm flag of state (AFS). Both have $\mathcal{B}$ attribute.

AD (Algorithm Data Area): these data help the specified algorithm execute. It has $\mathcal{D}$ attribute.

MF (Message Flag Area): includes defined message flag (DMF) and user-customized message flag (UMF). DMF is the necessary message for system execution, e.g., start module flag, alarm flag, etc. UMF can be defined by users. Both have $\mathcal{B}$ attribute.

MD (Message Data Area): used to transfer message information, which includes system message data area (DMD) and user message data area (UMD). It is defined in D area.

MPDIF (Master Processor Data Interaction Flag Area): contains begin data transfer flag from master to slave (MSB), transfer state of master from master to slave (MSF), acknowledge flag of master from master to slave (MSA), and transfer state of master from slave to master (MSS). All have $\mathcal{B}$ attribute.

MSD (Master Processor Data Interaction Data Area): an area that stores the data delivered from slave processors. It has $\mathcal{D}$ attribute.

SPDIF (Slave Processor Data Interaction Flag Area): includes the begin data transfer flag from slave to master (SMB), transfer state of slave from slave to master (SMF), acknowledge flag of slave from slave to master (SMA), and transfer state of slave from master to slave (SMS). All have $\mathcal{B}$ attribute.

SMD (Slave Processor Data Interaction Data Area): stores the data delivered from master processor. It has $\mathcal{D}$ attribute.

D. *User-oriented Thread Design*

From the user's point of view, in most cases, the logic control program (LCP) and algorithm program (AP) can be developed independently [**?**]; hence, we have a logic thread and algorithm thread. However, in order to satisfying the ever-growing performance requirements of users, we proposed the customized multiprocessor $ePLC$. Correspondingly, an individual motion thread is designed into every slave processor.

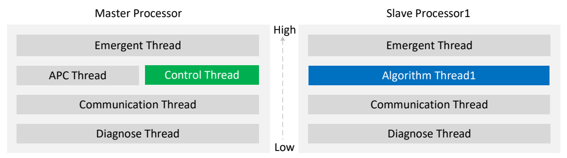


Fig. 5. User-oriented thread design in master and slave processors.

The user-oriented thread structure can be seen in Fig 5. Every processor is a four-level pre-emptive scheduling thread structure: **Emergent Thread**, **Communication Thread**, **Diagnose Thread**, and **APC Thread** as seen in [**?**]. Two special threads are explained below.

Control Thread: runs in the master processor and has functions including dealing with DI\O, executing logic program, exchanging data with slave processors, etc.

Algorithm Thread: runs in the slave processor and contains functions including interacting data with master, executing algorithm program, controlling actuators, etc.

E. *Formal Description of Software Structure*

Normally, the software structure could be divided into two-layer structure: a control layer for the logic program and a motion layer for the motion control. In addition, considering the advantage of complexity reduction in modular design [13], we therefore provide a system-level frame for modular design. Hence, the software structure with modular design is depicted in Fig. 3. The program is composed of many modules, and a module consists of a logic program and several related algorithms. Modules work under mechanisms: reasonable memory allocation, LPM interaction, with a data mechanism, and running multi-threading. For a clearer description, we can define an $ePLC$ program as follows:

$$\begin{cases} PS = \{MS, MA, LPM, TD\}, \\ ms_i \in MS = \{lcp_i, \bigcup_{j=g}^{h} ap_j\}, \\ lcp_i \in LCP = \{ip_i, lcf_i, lcd_i, lpb_i\}, \\ ap_j \in AP = \{afe_j, afs_j, ad_j, ab_j\}. \end{cases} \tag{1}$$

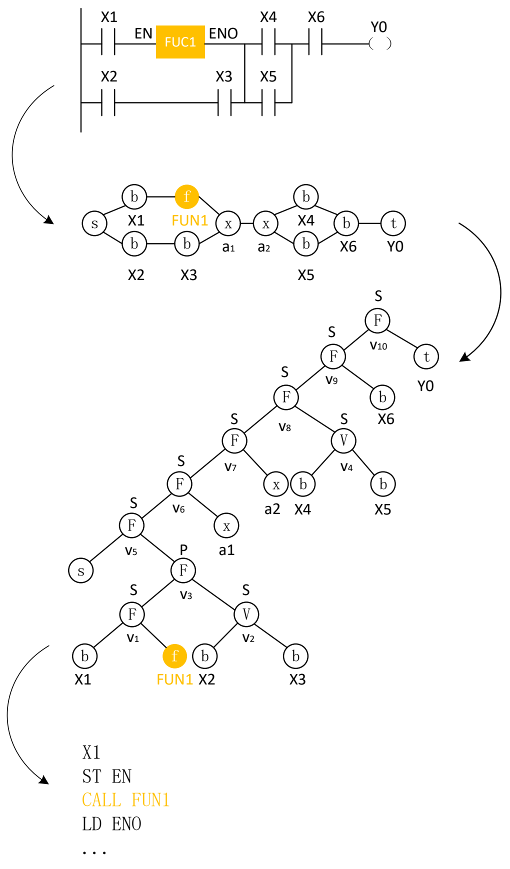
The programming structure (PS) consists of modules (MS), memory allocation (MA), LPM data interaction, and threads (TD). Each ms_i has two parts: a logic control program lcp_i and several algorithm programs ($ap_g, ap_{g+1}, ..., ap_j, ..., ap_h$). Each lcp_i includes an initial program (ip_i), logic control flag (lcf_i), logic control data (lcd_i), and logic program body (lpb_i). Each ap_j contains afe_j, afs_j, ad_j, and algorithm body (ab_j).

III. System Implementation

A. *Compilation of graphic program*

The compilation of the graphic program contains two parts: 1) compiling the graphic language into instruction list, and 2) compiling the multi-language components.

In the first part, since we see the multi-language components as a common component, the compilation of graphic language embedded multi-language components is almost the same as the process of [36], in which readers can find a detailed



```
X1
ST EN
CALL FUN1
LD ENO
...
```

Fig. 6. Three steps in compilation of graphic program, which contains multi-language component: 1) convert topology structure to directed graph, 2) generate a binary decomposition tree, and 3) generate IL instructions.

explanation. As shown in Fig. 6, we adopt three steps to implement the first of the two above-mentioned compilations:

Step 1: Convert topology structure to directed graph according to LD syntax library and analyze the errors of the topology.

Step 2: Generate a binary decomposition tree according to series and parallel rules.

Step 3: Generate IL instructions according to the IL grammar library. For multi-language components, it is described as a program entry.

In the second part, the multi-language components are compiled. For the convenience of users, they can still use the same grammar to program the $ePLC$ dedicated storage area inside the multi-language component, such as $M2000 = 1$, which represents giving 1 to the bit area $M2000$, whereas it is illegal in other languages. Hence, we should compile the component to the identifiable code which contains the following two steps:

Step 1: Address mapping. Every type of processor has its own address mapping rules (AMR), e.g.,

$$APR = \{CID, MAS, DAS, \mathcal{A}_m, \mathcal{A}_d\}, \tag{2}$$

where CID is the compiler identity and MAS and DAS are the start address of M and D areas, respectively. $\mathcal{A}_m$ and $\mathcal{A}_d$ are the rules for mapping M and D to the address of the processor, respectively, see Algorithms 1 and2. Algorithm 1 translates the four operators of each element m_i, which are $\mathcal{S}_0(m_i)$, $\mathcal{S}_1(m_i)$, $\mathcal{J}_0(m_i)$, and $\mathcal{J}_1(m_i)$, to recognizable the form of the CID compiler. In addition, in the PLC platform, we adopt the octal system, so it is necessary to translate the octal number to a decimal number. Algorithms 1 and 2 both contain this process.

Step 2: Call the corresponding compiler to compile the component.

Algorithm 1: $\mathcal{A}_m$

```
Input: string oStr of $m_i$ contained its operator
Output: converted string cStr
Get the number $i$ from oStr;
Get operator $opt$ from oStr;
remainder r = $i\%10$;
Octal oI = $i/10$;
Convert oI to decimal dI;
for opt do
    if opt==$\mathcal{S}_0$ then
        cStr = "CassMen[$MAS$+dI] $\gg$ r == 0";
    end
    if opt==$\mathcal{S}_1$ then
        cStr = "CassMen[$MAS$+dI] $\gg$ r == 1";
    end
    if opt==$\mathcal{J}_0$ then
        cStr = "CassMen[$MAS$+dI] |~(1 $\ll$ r)";
    end
    if opt==$\mathcal{J}_1$ then
        cStr = "CassMen[$MAS$+dI] & (1 $\ll$ r)";
    end
end
```

Algorithm 2: $\mathcal{A}_d$

```
Input: string oStr of $d_i$
Output: converted string cStr
Get the number $i$ from oStr;
Convert $i$ to decimal dI;
cStr = "CassMen[$DAS$+dI]";
```

B. LPM data interaction

In order to implement data interaction among different parts and processors, we propose the LPM-data-interaction method. As shown in Fig. 7, we define the LPM data interaction as having three parts: $\mathcal{L}$ (layer data interaction), $\mathcal{P}$ (processor data interaction), and $\mathcal{M}$ (module data interaction). $\mathcal{L}$ seen in [**?**] is the process of exchanging the data between the application-customized layer and control layer.

$\mathcal{P}$ is used to establish the data interaction between master processor and slave processors; hence, it contains the process of transferring data from master to slave ($\mathcal{P}_{mts}$) and the

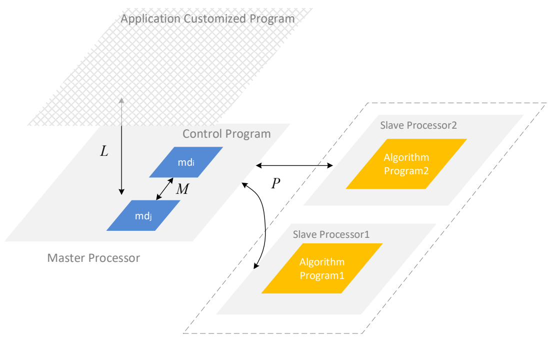


Fig. 7. LPM data interaction defined as having three parts: $\mathcal{L}$ (layer data interaction), $\mathcal{P}$ (processor data interaction), and $\mathcal{M}$ (module data interaction).

process of transferring data from slave to master ($\mathcal{P}_{stm}$), both of which are defined as follows:

$$\begin{cases} \mathcal{P}_{mts} = \mathcal{U}(msb_i, msf_i, sma_i, sms_i, smd_i), \\ \mathcal{P}_{stm} = \mathcal{U}(smb_i, smf_i, msa_i, mss_i, msd_i), \end{cases} \tag{3}$$

where $\mathcal{U}$ is the function that implements the process of data interaction between master and slave processors. $\mathcal{P}_{mts}$ and $\mathcal{P}_{mts}$ use the same function $\mathcal{U}$.

The process flow of $\mathcal{P}_{mts}$ is expressed as follows: $\mathcal{S}_1(msb_i) \rightarrow \mathcal{S}_1(msf_i) \rightarrow send(smd_i) \rightarrow \mathcal{S}_0(msb_i) \rightarrow \mathcal{S}_1(sms_i) \rightarrow check(smd_i) \rightarrow \mathcal{S}_1(sma_i) \rightarrow \mathcal{S}_0(sms_i) \rightarrow \mathcal{S}_0(sma_i) \rightarrow \mathcal{S}_0(msf_i)$.

Here, $send(msd_i)$ denotes sending data to msd_i in the slave processor. $check(msd_i)$ denotes checking the data of msd_i. $\rightarrow$ denotes the transition to the next step, e.g., $\mathcal{S}_1(msb_i) \rightarrow \mathcal{S}_1(msf_i)$ denotes that msb_i and is set one, and then msf_i is set one.

$\mathcal{M}$ is used to establish the data interaction among modules. It includes two types of messages: system-defined message interaction $\mathcal{M}_d$ and user message interaction $\mathcal{M}_u$. The process is defined as follows:

$$\begin{cases} \mathcal{M}_d = \mathcal{V}(dmf_i, dmd_i, \mathcal{E}), \\ \mathcal{M}_u = \mathcal{V}(umf_i, umd_i\, \mathcal{E}), \end{cases} \tag{4}$$

where $\mathcal{V}$ is the function used to broadcast messages and transfer data, $\mathcal{E}$ is the collection of all execution functions after receiving a related message, and $\mathcal{M}_s$ and $\mathcal{M}_u$ have the same function $\mathcal{V}$.

One module receives a system-defined message as follows: $\mathcal{J}_1(smf_i) \rightarrow GetMessage_j(dmf_i) \rightarrow GetData(dmd_i) \rightarrow \mathcal{E}_j$.

Here, $GetMessage_j(dmf_i)$ represents the ith module receiving a message dmf_i and $GetData(dmd_i)$ represents the ith module receiving the message information.

C. Execution of threads

Commonly, threads in master processor and in slave processors execute separately according to their priority, and the interaction between control thread and algorithm threads occurs when using $\mathcal{P}_{mts}$ and $\mathcal{P}_{mts}$. Basic execution units of control thread are as follows:

C_1: start a module.
C_2: transfer data to slave processor by $\mathcal{P}_{mts}$.
C_3: deal with the feedback data.
C_4: broadcast a message.
C_5: handle the message.

A motion thread contains the following basic execution units:

M_1: start the algorithm.
M_2: execute the algorithm.
M_3: feedback the data to master processor by $\mathcal{P}_{stm}$.
M_4: end algorithm.

Two cases shown in Fig. 8 are explained below:

Case 1: Execution of a control thread and two algorithm threads among three processors. The control thread (CT) traverses LCF, finds ms_i to be executed , runs lcp_i, finds ap_j, executes C_1 unit, executes C_2 unit, and then transfers data from LCD to SMD of processor 1. Algorithm thread (AT) 1 executes M_1 unit, executes M_2 after transferring data from SMD to AD, runs M_3 unit, and feedbacks data to CT. When ap_j finishes, AT 1 executes M_4 and informs CT of the end of ap_j. CT executes C_3 to end the process and then finds ap_{j+1}, executes C_1 and C_2, transfers the data from LCD to SMD of processor 2, AT 2 executes M_1 unit, and executes M_2 after transferring data from SMD to AD. When ap_{j+1} finishes, AT 2 executes M_4 unit and informs CT of the end of ap_{j+1}. CT executes C_3 to finish the process.

Case 2: Execution of control thread and two algorithm threads among three processors together with message mechanism. The CT traverses LCF, finds ms_i to be executed , runs lcp_i, finds ap_j, executes C_1 unit, executes C_2 unit, and then transfers data from LCD to SMD of processor 1. AT 1 executes M_1 unit and executes M_2 after transferring data from SMD to AD. During the execution of ms_i, CT executes C_4 to broadcast the message dmf_x to inform ms_k to run. After execution of C_5, ms_k obtains data and starts, and then CT finds ap_{j+1} in ms_k, executes C_1 and C_2, transfers the data from LCD to SMD of processor 2, AT 2 executes M_1 unit, and executes M_2 after transferring data from SMD to AD. During the execution of ap_{j+1}, AT 1 executes M_4 and informs CT to execute C_3. After that, ap_{j+1} finishes, and then CT executes C_3 to finish the process.

D. Finite state machines

The finite state machines adopt the 5-tuple, which is similar to [37]:

$$\mathcal{F} = (Q, X, Y, \delta, \lambda), \tag{5}$$

where $Q = \{q_0, q_1, ..., q_i\}$ is the collection of states, $q_0 \in Q$ is the initial state, X is the finite set of inputs, Y is the finite set of outputs, and δ is the state transition function $\delta : Q \times X \rightarrow Q$. λ is the output function $\lambda : Q \times X \rightarrow Y$. If F is in state q and x occurs, then F transitions to state $q' = \delta(q, x)$ and outputs $y = \lambda(q, x)$. This transition is denoted $\tau = (s, x/o, s')$.

Hence, we have the following finite state machines of the master processor:

$$\begin{cases} \mathcal{F}_m = \{Q_m, X_m, Y_m, \delta_m, \lambda_m\}, \\ Q_m = \{mstop, mrun, mbm, pdi\}, \\ X_m = \{x_{m1}, x_{m2}, x_{m3}, x_{m4}, x_{m5}, x_{m6}, x_{m7}\}, \\ Y_m = \{y_{m1}, y_{m2}, y_{m3}\}, \end{cases} \tag{6}$$

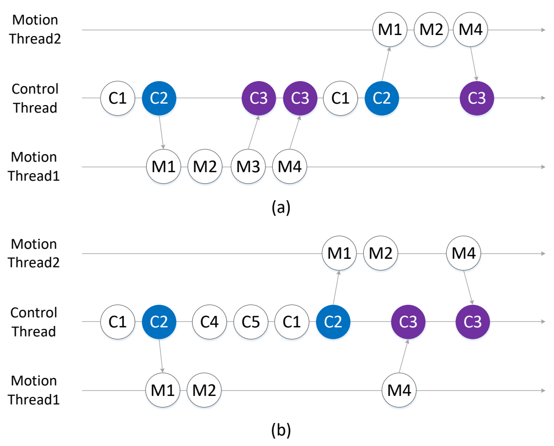


Fig. 8. Execution of control thread and two algorithm threads among three processors with and without messages.

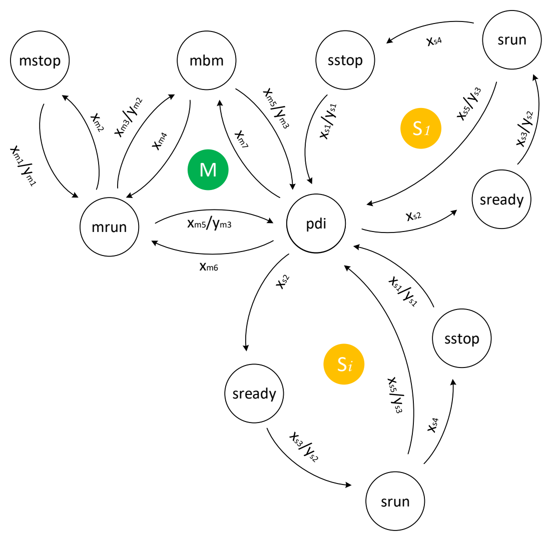


Fig. 9. Finite state machines of $ePlC$, which contains master processor M, slave processor S_1, and slave processor S_i.

where $mstop$ is the stop state and the initial state of master processor, $mrun$ is the run state, mbm is the broadcast state, and pdi is the data interaction state. The inputs are defined as follows:

$$\begin{aligned}
&x_{m1} \Leftrightarrow \exists lcf_i : \mathcal{J}_1(lcf_i),\\
&x_{m2} \Leftrightarrow \forall lcf_i : \mathcal{J}_0(lcf_i),\\
&x_{m3} \Leftrightarrow \exists mf_i : \mathcal{J}_1(mf_i),\\
&x_{m4} \Leftrightarrow \forall mf_i : \mathcal{J}_0(mf_i),\\
&x_{m5} \Leftrightarrow \exists msf_i, mss_j : \mathcal{J}_1(msf_i) \vee \mathcal{J}_1(mss_j),\\
&x_{m6} \Leftrightarrow \forall msf_i, mss_j, mf_k : \mathcal{J}_0(msf_i) \wedge \mathcal{J}_0(mss_j) \wedge \mathcal{J}_0(mf_k),\\
&x_{m7} \Leftrightarrow \forall msf_i, mss_j, \exists mf_k : \mathcal{J}_0(msf_i) \wedge \mathcal{J}_0(mss_j) \wedge \mathcal{J}_1(mf_k),\\
&y_{m1} \Leftrightarrow C_1,\\
&y_{m2} \Leftrightarrow C_4,\\
&y_{m3} \Leftrightarrow C_2 \quad or \quad C_3.
\end{aligned}$$

Here, e.g., $x_{m1} \Leftrightarrow \exists lcf_i : \mathcal{J}_1(lcf_i)$ denotes that x_{m1} is an input of X_m and this input is equivalent to the existence of a logic control flag lcf_i whose value is 1.

We than can obtain the state transitions of the master processor:

$$\begin{cases}
\tau_m 1 = (mstop, x_{m1}/y_{m1}, mrun),\\
\tau_m 2 = (mrun, x_{m2}, mstop),\\
\tau_m 3 = (mrun, x_{m3}/y_{m2}, mbm),\\
\tau_m 4 = (mbm, x_{m4}, mrun),\\
\tau_m 5 = (mrun, x_{m5}/y_{m3}, pdi),\\
\tau_m 6 = (pdi, x_{m6}, mrun),\\
\tau_m 7 = (mbm, x_{m5}/y_{m3}, pdi),\\
\tau_m 8 = (pdi, x_{m7}, mbm).
\end{cases} \tag{7}$$

The finite state machines of every slave processor are as follows:

$$\begin{cases}
\mathcal{F}_s = \{Q_s, X_s, Y_s, \delta_s, \lambda_s\},\\
Q_s = \{sstop, sready, srun, pdi\},\\
X_s = \{x_{s1}, x_{s2}, x_{s3}, x_{s4}, x_{s5}\},\\
Y_s = \{y_{s1}, y_{s2}, y_{s3}\},
\end{cases} \tag{8}$$

where $sstop$ is the stop state and the initial state, $srun$ is the run state, $sready$ is the ready state, and pdi is the data interaction state. The events are defined as follows:

$$\begin{aligned}
&x_{s1} \Leftrightarrow \exists sms_i : \mathcal{J}_1(sms_i),\\
&x_{s2} \Leftrightarrow \exists afe_i : \mathcal{J}_1(afe_i),\\
&x_{s3} \Leftrightarrow \exists afs_i : \mathcal{J}_1(afs_i),\\
&x_{s4} \Leftrightarrow \forall afs_i : \mathcal{J}_0(afs_i),\\
&x_{s5} \Leftrightarrow \exists smb_i, sms_j : \mathcal{J}_1(smb_i) \vee \mathcal{J}_1(sms_j),\\
&y_{s1} \Leftrightarrow M_1,\\
&y_{s2} \Leftrightarrow M_2,\\
&y_{s3} \Leftrightarrow M_3.
\end{aligned}$$

We then can obtain the state transitions of the slave processor:

$$\begin{cases}
\tau_{s1} = (sstop, x_{s1}/y_{s1}, pdi),\\
\tau_{s2} = (pdi, x_{s2}, sready),\\
\tau_{s3} = (sready, x_{s3}/y_{s2}, srun),\\
\tau_{s4} = (srun, x_{s4}, sstop),\\
\tau_{s5} = (srun, x_{s5}/y_{s3}, pdi).
\end{cases} \tag{9}$$

Figure 9 illustrates all of the finite state machines of $ePLC$, which contains master processor M, slave processor S_1, and slave processor S_i. This is the graphical form of Eqs. 6 and 8.

IV. Experiment

A. Distributed control system

As shown in Fig. 10, we verified the proposed development method on a 200-T IMM. The TI F28M35 chip was chosen as the main chip of the $ePLC$. It has two cores: a TI C28x and an ARM Cortex M3. Considering the DSP is more suitable for motion control, the Cortex M3 is chosen as the master processor and C28x as the slave processor. The $ePLC$ has an RS232 and a CAN. The RS232 is used to download programs and connect with the HMI, and the CAN is designed to extend the DI\O and AI\O. The HMI can be customized by users.

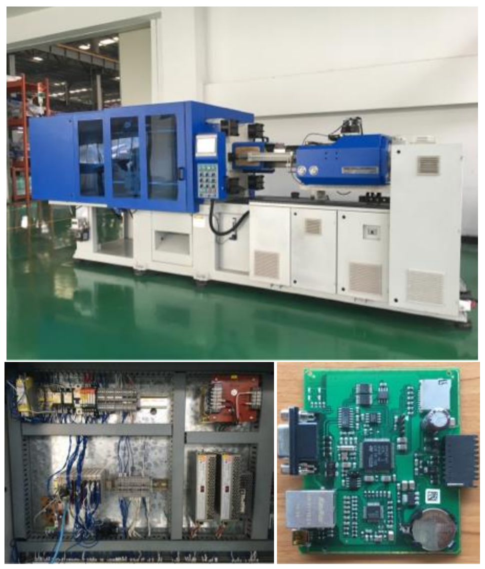
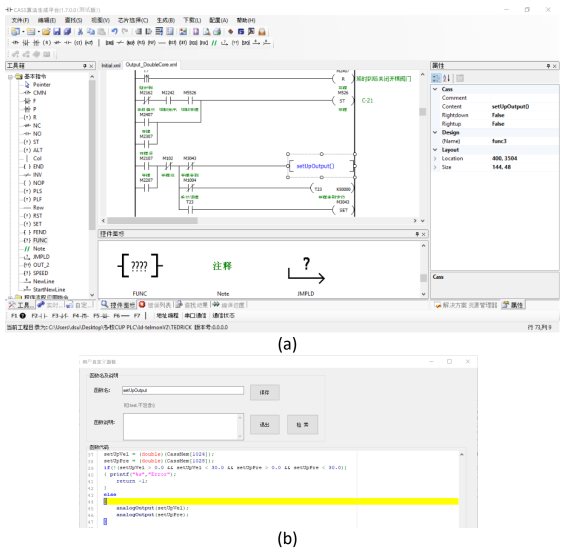


Fig. 10. 200-T IMM and multiprocessor $ePLC$.



setUpOutput

```
setUpVel = (double)(CassMem[1024]);
setUpPre = (double)(CassMem[1028]);
if(!(setUpVel > 0.0 && setUpVel < 30.0 && setUpPre > 0.0 && setUpPre < 30.0))
{ printf("%s","Error");
    return -1;
}
else
{
    analogOutput(setUpVel);
    analogOutput(setUpPre);
}
```

Fig. 11. (a) uniform development platform with C-language component in dotted-line box and (b) its partial code. This component represents to output a small velocity and pressure in setup mode.

B. Software structure

Figure 11 shows the uniform development platform that we devised. The dotted-line box of Fig. 11 (a) represents the C-language component and Fig. 11 (b) its partial code. This component represents to output a small velocity and pressure in setup mode.

After the design of every used component, we designed the modules according to Table I. Additionally, a special module was designed to control the execution flow of all modules with the DMF and DMD.

Three typical requirements for using the proposed user-oriented development method are as follows:

1) Using multiprocessors: Add S-curve acceleration and deceleration algorithms. For this case, we ran the S-curve in the DSP (slave processor).
2) Design in the uniform development platform: Add an ejector module containing a T-curve. We designed the LCP with LD and the T-curve packaged as a component in the same platform.
3) Modular design: Inject before high pressure of the mold closes. We customized a message flag in MSF and broadcast the message at the beginning of high pressure, after which the injection module received the message and the CT started it.

C. Analysis

We compared the system information, development method, and key performance indicators among the TECHMATION system, the KEBA system, and the proposed system. The compared controller types of the KEBA and TECHMATION systems are i1000 and TE-CH2, respectively.

1) System information. The TECHMATION and KEBA systems account for the majority of market share. As shown in Table III, the KEBA and TECHMATION systems have more powerful CPUs and more RAM than the implemented system. The main frequency of the KEBA and TECHMATION systems are 400 and 80 MHz, respectively, and their RAM 128 MB and 512 kB, respectively. In contrast, the main frequency and RAM of the implemented system are 72 MHz and 132 kB, respectively. Owing to the reduced complexity, our system can have a customized PLC and separate HMI. To some extent, this lowers the cost significantly. The proposed system adopts the widely used distributed structure, which decreases wire usage and increases resistance to interference.
2) Development method. Our system adopted a user-oriented development method, including a customized multiprocessor $ePLC$, component-based uniform development platform, and comprehensive optimization of the system. It is difficult to achieve these in other systems and they cannot support a C-language component. In particular, the TECHMATION system is developed by assembly instruction and does not support IEC61131-3.
3) Key performance indicators. With our best knowledge, we adopted defective percentage (DP), error of change-over position (ECP), error of cushion minimum (ECM), error of charging end position ($ECEP$), and error of mold open end position ($EMOEP$) as the key performance indicators. All the systems were adjusted to use a T-curve and the key parameters were set to the same value. The cycle time, mold close time, mold open time, injection time, charging and cooling time, and ejector forward time and ejector backward time were controlled at approximately 8, 2, 2, 1, 1, 1, 0.5, and 0.5 s, respectively. Figure 12 shows the 100-times error line graph of the key performance indicators. The proposed system has nearly identical performance as

TABLE III
SYSTEM INFORMATION COMPARISON

	Controller model	CPU	Main frequency	RAM
KEBA	Series i1000	PowerPC	400 MHz	128 MB
TECHMATION	TE-CH2	DSP54	80 MHz	512 kB
Implemented	-	F28M35	72 MHz[1]	132 kB[2]

[1] Main frequencies of ARM and DSP are 100 and 150 MHz, respectively.
[2] Value is the summation of RAM, DSP, and shared RAM.

TABLE IV
KEY PERFORMANCE COMPARISON

Brand	ST	DP	$\overline{ECP}$ [3]	$\overline{ECM}$	$\overline{ECEP}$	$\overline{EMOEP}$
TECHMATION	27 s	0.27%	0.094	0.2	0.13	0.17
KEBA	72 s	0.24%	0.064	0.1	0.1	0.12
Implemented	5 s	0.25%	0.05	0.1	0.11	0.11

[3] $\overline{ECP}$ denotes the 100-time mean of ECP.

the KEBA system, which is better than that of the TECHMATION system. Table I shows the comparison of startup time (ST), DP, and the mean of the key performance indicators. The proposed system's startup time was increased by more than 5 times in the case of almost identical key performance.

V. CONCLUSIONS

In this paper, we presented a user-oriented development method in which a customized multiprocessor $ePLC$ was proposed to enhance performance, a multi-language-supporting graphical component was proposed to improve the adaptability of developers, and an optimized system structure (reasonable memory allocation, user-oriented thread structure, LPM data interaction, modular software design, and finite state machines) was proposed to reduce the developmental complexity. Ultimately, we adopted the proposed method to implement a distributed IMM system. Through comparison with the TECHMATION and KEBA systems, our system startup time increased by more than 5 times in the case of almost identical key performance. Furthermore, our system supports a customized multiprocessor $ePLC$ and detached HMI.

In planned additional work, more applications will be implemented to prove the system's robustness. Meanwhile, integration with other systems (e.g., a visual system) will be undertaken to study the system flexibility.

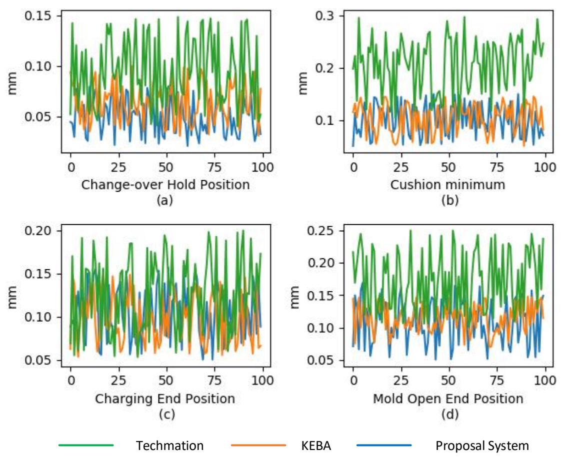


Fig. 12. (a) is ECP, (b) is ECP, (c) is ECM, (d) is EMOEP.

REFERENCES

[1] D. A. Chekired, L. Khoukhi, and H. T. Mouftah, "Industrial iot data scheduling based on hierarchical fog computing: A key for enabling smart factory," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–1, 2018.

[2] Y. Jiang, H. Zhang, H. Liu, X. Song, W. N. Hung, M. Gu, and J. Sun, "System reliability calculation based on the run-time analysis of ladder program," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013, pp. 695–698.

[3] Y. Jiang, H. Zhang, X. Song, X. Jiao, W. N. N. Hung, M. Gu, and J. Sun, "Bayesian-network-based reliability analysis of plc systems," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 11, pp. 5325–5336, 2013.

[4] B. F. Adiego, D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized plc programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[5] S. Gerkšič, G. Dolanc, D. Vrančić, J. Kocijan, S. Strmčnik, S. Blažič, I. Škrjanc, Z. Marinšek, M. Božiček, and A. Stathaki, "Advanced control algorithms embedded in a programmable logic controller," *Control Engineering Practice*, vol. 14, no. 8, pp. 935–948, 2006.

[6] C. Y. Chang, "Adaptive fuzzy controller of the overhead cranes with nonlinear disturbance," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 2, pp. 164–172, 2007.

[7] S. Dominic, Y. Lohr, A. Schwung, and S. X. Ding, "Plc-based real-time realization of flatness-based feedforward control for industrial compression systems," *IEEE Transactions on Industrial Electronics*, vol. PP, no. 99, pp. 1–1, 2016.

[8] D. Schütz, A. Wannagat, C. Legat, and B. Vogel-Heuser, "Development of plc-based software for increasing the dependability of production automation systems," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2397–2406, 2013.

[9] J. D. L. Morenas, P. G. Ansola, and A. García, "Shop floor control: A physical agents approach for plc-controlled systems," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–1, 2017.

[10] M. F. Zaeh, C. Poernbacher, and J. Milberg, "A model-based method to develop plc software for machine tools," *CIRP Annals - Manufacturing Technology*, vol. 54, no. 1, pp. 371–374, 2005.

[11] S. Hossain, M. A. Hussain, and R. B. Omar, "Advanced control software framework for process control applications," *International Journal of Computational Intelligence Systems*, vol. 7, no. 1, pp. 37–49, 2014.

[12] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Engineering Practice*, vol. 21, no. 11, pp. 1608–1619, 2013.

[13] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.

[14] X. M. Shi, W. J. Fei, and S. P. Deng, "The research of circular interpolation motion control based on rectangular coordinate robot," *Key Engineering Materials*, vol. 693, pp. 1792–1798, 2016.

[15] N. Fang, "Design and research of multi axis motion control system based on plc," *Academic Journal of Manufacturing Engineering*, vol. 15, no. 1, pp. 17–23, 2017.

[16] W. F. Peng, G. H. Li, P. Wu, and G. Y. Tan, "Linear motor velocity and acceleration motion control study based on pid+velocity and acceleration feedforward parameters adjustment," *Materials Science Forum*, vol. 697-698, pp. 239–243, 2011.

[17] A. Syaichu-Rohman and R. Sirius, "Model predictive control implementation on a programmable logic controller for dc motor speed control," in *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*. IEEE, 2011, pp. 1–4.

[18] J. Qian, H. B. Zhu, S. W. Wang, and Y. S. Zeng, "A 5-dof combined robot platform for automatic 3d measurement," *Key Engineering Materials*, vol. 579-580, pp. 641–644, 2014.

[19] P. Co.Ltd, *Programmable controller FP2 Positioning Unit Mannual*, 2011.

[20] M. Dubois and C. Scheurich, “Memory access dependencies in shared-memory multiprocessors,” *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 660–673, 2002.

[21] J. H. Patel, “Processor-memory interconnections for multiprocessors,” *IEEE Transactions on Computers*, vol. C-30, no. 10, pp. 771–780, 2006.

[22] D. Zhu, L. Chen, S. Yue, T. Pinkston, and M. Pedram, “Providing balanced mapping for multiple applications in many-core chip multiprocessors,” *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 3122–3135, 2016.

[23] L. M. Albarakat, P. V. Gratz, and D. A. Jiménez, “Mtb-fetch: Multithreading aware hardware prefetching for chip multiprocessors,” *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2017.

[24] Z. Hajduk, B. Trybus, and J. Sadolewski, “Architecture of fpga embedded multiprocessor programmable controller,” *IEEE Transactions on Industrial Electronics*, vol. 62, no. 5, pp. 2952–2961, 2015.

[25] M. Chmiel, J. Kulisz, R. Czerwinski, A. Krzyzyk, M. Rosol, and P. Smolarek, “An iec 61131-3-based plc implemented by means of an fpga,” in *IEEE/IFAC PDES Conference*, 2016, pp. 28–37.

[26] O. Co.Ltd, “Cs1w-mc221(-v1)/mc421(-v1) motion control units.” *Operation Mannual*, 2004.

[27] I. 61131-3, *Programmable controllers - part 3: Programming languages.*, 1993.

[28] P. T. C. 2., *Function blocks for motion control version 1.1*, 2005.

[29] C. Sünder, A. Zoitl, F. Mehofer, and B. Favre-Bulle, “Advanced use of plcopen motion control library for autonomous servo drives in iec 61499 based automation and control systems,” *E & I Elektrotechnik Und Informationstechnik*, vol. 123, no. 5, pp. 191–196, 2006.

[30] S. S. S. GmbH, “Logic and motion control integrated in one iec 61131-3 system:development kit for convenient engineering of motion, cnc and robot applications.” 2017.

[31] M. Bonfe and C. Fantuzzi, “Object-oriented approach to plc software design for a manufacture machinery using iec 61131-3 norm languages,” in *Ieee/asme International Conference on Advanced Intelligent Mechatronics, 2001. Proceedings*, 2001, pp. 787–792 vol.2.

[32] B. Werner, “Object-oriented extensions for iec 61131-3,” *Industrial Electronics Magazine IEEE*, vol. 3, no. 4, pp. 36–39, 2009.

[33] F. Basile, P. Chiacchio, and D. Gerbasio, “On the implementation of industrial automation systems based on plc,” *IEEE Transactions on Automation Science & Engineering*, vol. 10, no. 4, pp. 990–1003, 2013.

[34] O. Verscheure, X. Garcia, G. Karlsson, and J. P. Hubaux, “User-oriented qos in packet video delivery,” *IEEE Network*, vol. 12, no. 6, pp. 12–21, 2016.

[35] N. Choi, D. Kim, S. J. Lee, and Y. Yi, “A fog operating system for user-oriented iot services: Challenges and research directions,” *IEEE Communications Magazine*, vol. 55, no. 8, pp. 44–51, 2017.

[36] Y. Yan and H. Zhang, “Compiling ladder diagram into instruction list to comply with iec 61131-3,” *Computers in Industry*, vol. 61, no. 5, pp. 448–462, 2010.

[37] R. M. Hierons and U. C. Turker, “Parallel algorithms for testing finite state machines: Generating uio sequences.” *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1077–1091, 2016.