Algorithm 1: C_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*==*0* **then**
    cStr = "CassMen[MAS+dI] $\gg$ r == 0";
  end
  if *opt*==*1* **then**
    cStr = "CassMen[MAS+dI] $\gg$ r == 1";
  end
  if *opt*==*2* **then**
    cStr = "CassMen[MAS+dI] |~(1 $\ll$ r)";
  end
  if *opt*==*3* **then**
    cStr = "CassMen[MAS+dI] & (1 $\ll$ r)";
  end
end

Algorithm 2: C_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]";

$\mathcal{P}_{mts}$ and $\mathcal{P}_{mts}$ have the same execution function $\mathcal{I}$ and the process is seen as below: $\mathcal{O}(ms_i) \rightarrow \mathcal{O}(mf_i) \rightarrow send(sd_i) \rightarrow \mathcal{O}(sf_i) \rightarrow check(sd_i) \rightarrow \mathcal{O}(sa_i) \rightarrow \mathcal{O}(mc_i) \rightarrow \mathcal{F}(ms_i) \rightarrow \mathcal{F}(mf_i) \rightarrow \mathcal{F}(mc_i) \rightarrow \mathcal{F}(sf_i) \rightarrow \mathcal{F}(sa_i)$.

where $send(sd_i)$ means sending data to sd_i in slave processor. $check(sd_i)$ means to check data of sd_i.

$\mathcal{M}$ is used to interact data among modules. It includes two types message: system defined message interaction $\mathcal{M}_s$ and user defined message interaction $\mathcal{M}_u$. The process is defined as below:

$$\begin{cases} \mathcal{M}_s = \mathcal{J}(smf_i, smd_i, \mathcal{C}) \\ \mathcal{M}_u = \mathcal{J}(umf_i, umd_i\, \mathcal{C}) \end{cases} \tag{5}$$

Where $\mathcal{C}$ is the collection of all callback functions, $\mathcal{M}_s$ and $\mathcal{M}_u$ have the same execution function $\mathcal{J}$ and one module receives a message shown as below: $\mathcal{O}(smf_i) \rightarrow GetMessage_j(smf_i) \rightarrow GetData(msd_i) \rightarrow C_j$.

Where $GetMessage_j(smf_i)$ represents the ith module getting a message smf_i and $GetData(msd_i)$ represents the ith module getting 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

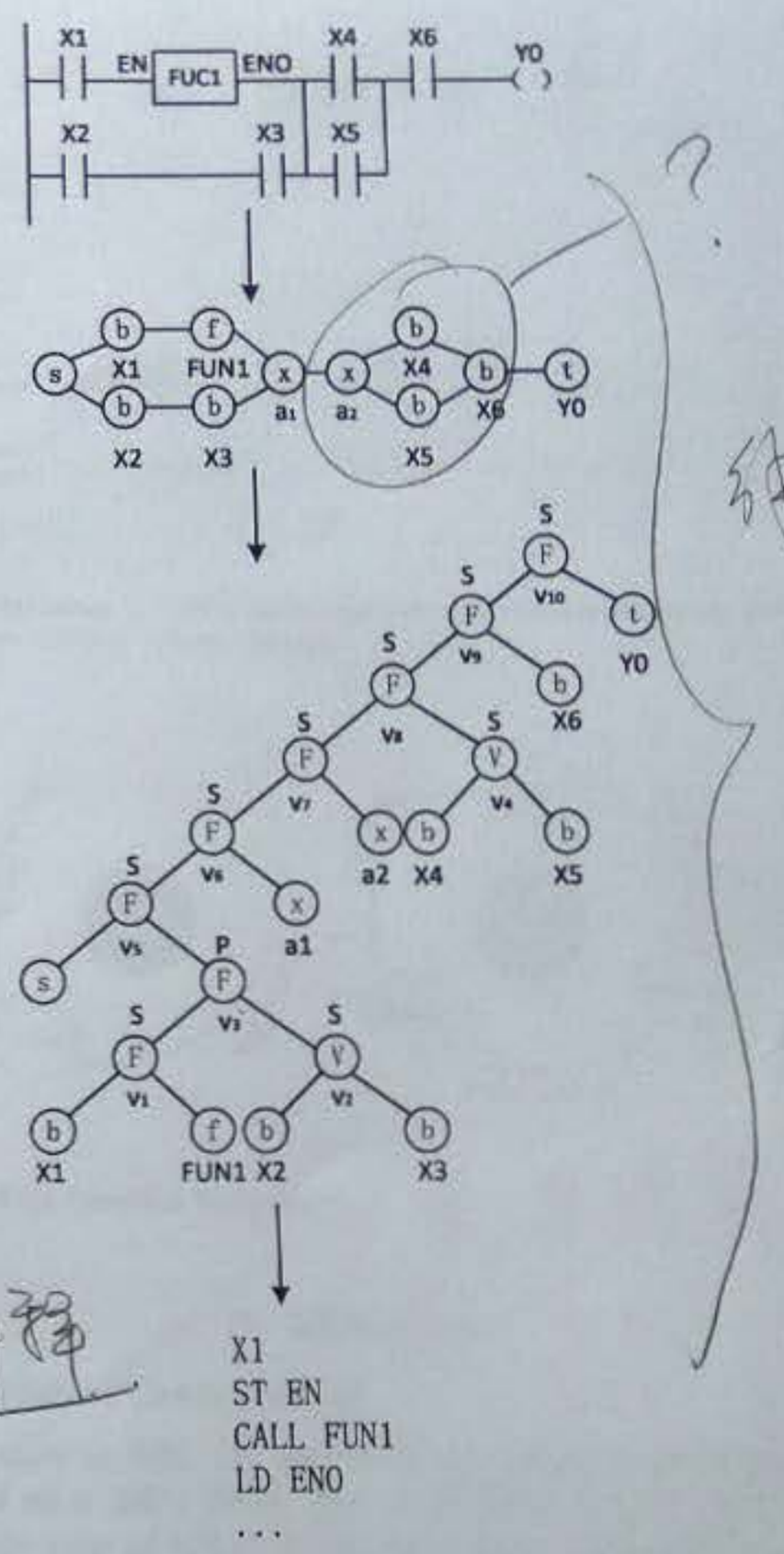

Fig. 6. Compilation of Graphic Program Which is Embedded Multi-language Components.

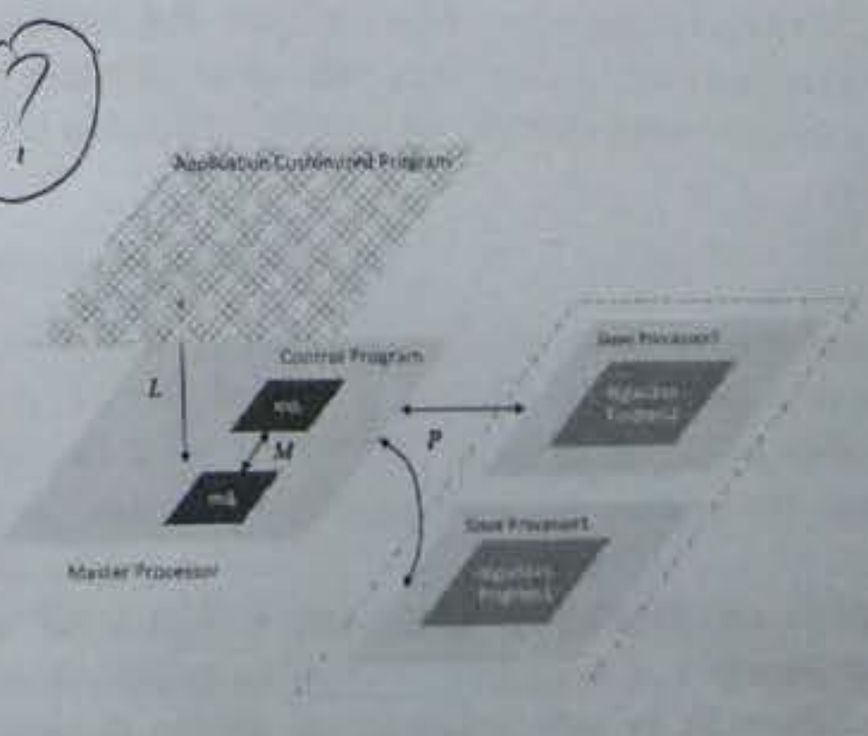

Fig. 7. LPM Data Interaction.

clear state flag in master (MC), transfer state of master from master to slave (MF), acknowledge flag of master from master to slave (MA) and transfer state of master from slave to master (MF). All of them have $\mathcal{B}$ attribute.

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

$SPDIF$ (Slave Processor Data Interaction Flag Area): this area includes the start data transfer flag from slave to master (SS), clear state flag in slave (SC) and transfer state of slave from slave to master (SF), acknowledge flag of slave from slave to master (SA) and transfer state of slave from master to slave (SF). All of them have $\mathcal{B}$ attribute.

SD (Slave Processor Data Interaction Data Area): an area stores the data delivered from master processor and 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) could be developed dividually [12], hence we have logic thread and algorithm thread. On the other hand, in order to satisfying ever-growing performance requirement of users, we proposed the customized multi-processor ePLC. Correspondingly an individual motion thread is designed into every slave processor. The user-oriented thread structure can be seen in FIG 5. Every processor is a four level preemptive scheduling thread structure. **Emergent Thread**, **Communication Thread**, **Diagnose Thread** and **APC Thread** see in [12]. Two special threads are explained as below.

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

Algorithm Thread: it is running in slave processor and contains functions including interacting data with master, executing algorithm program, control actuators, etc.

E. Modular Design

In applications, modular design will reduce the complexity of program. Therefore, we provide a system level frame for modular design. In FIG. 3, we can have a look on the modular design. The program is composed by a lot of modules and a module consists of logic program and several related algorithms. Modules will work under the reasonable memory allocation, data interaction mechanism and the running multithreading. Hence, we could define a program of complex logic control and motion control mixed as follows.

$$\begin{cases} PS = \{MD, MA, LPM, TD\} \\ md_i \in MD = \{lcp_i, \bigcup_{j=1}^{h} ap_j\} \\ lcp_i \in LCP = \{ip_i, lcf_i, lpb_i\} \\ ap_j \in AP = \{afe_j, afs_j, ad_j, ab_j\} \end{cases} \tag{2}$$

The programming structure (PS) consists of modules (MD), memory allocation (MA), LPM data interaction and threads (TD). Every MD_i has two parts: logic control program (LCP) and several algorithm programs (AP). LCP includes initial program (IP), LCF and logic program body (LPB). AP contains AFE, AFS, AD and algorithm body (AB).

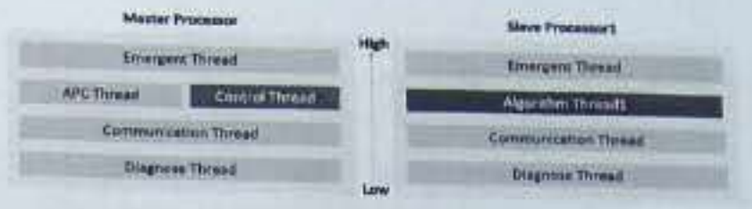

Fig. 5. User-oriented Thread Design in Master and Slave Processors.

III. System Implementation

A. Compilation of the graphic program

The compilation contains two parts: compiling graphic language into instruction list and compiling the multi-language components. As an example shown in FIG. 6, the compilation of graphic language embedded multi-language components is almost the same with the method in [36].

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

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

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

For the convenience of users, they still can use the same grammar to program the ePLC dedicated storage area inside multi-language component, such as $M2000$, whereas it is illegal in other languages. Hence, we should compile the component to the identifiable code which contains two steps.

Step 1 Address mapping. Every type of processor has its own AMR (Address Mapping Rules).

$$APR = \{CID, MAS, DAS, \mathcal{C}_m, \mathcal{C}_d\} \tag{3}$$

Where CID is the compiler identity, MAS and DAS are the start address of M and D area defined in processor respectively, $\mathcal{C}_m$ and $\mathcal{C}_d$ are the rules to map the M and D to the address of processor respectively. $\mathcal{C}_m$ and $\mathcal{C}_d$ see Algorithm 1 and Algorithm 2 respectively.

Step 2 Call the corresponding compiler to compile the component.

B. LPM data interaction

As shown in FIG. 7, we define the LPM data interaction with three parts: $\mathcal{L}$ (layer data interaction), $\mathcal{P}$ (processor data interaction) and $\mathcal{M}$ (module data interaction). $\mathcal{L}$ seen in [12] is the process to exchange the data between application customized layer and control layer.

$\mathcal{P}$ is used to interact data between master processor and slave processors, hence it has transferring data from master to slave ($\mathcal{P}_{mts}$) and transferring data from slave to master ($\mathcal{P}_{stm}$) and it is defined as below:

$$\begin{cases} \mathcal{P}_{mts} = \mathcal{I}(ms_i, mc_i, mf_i, sa_i, sf_i, sd_i) \\ \mathcal{P}_{stm} = \mathcal{I}(ss_i, sd_i, sf_i, ma_i, mf_i, md_i) \end{cases} \tag{4}$$

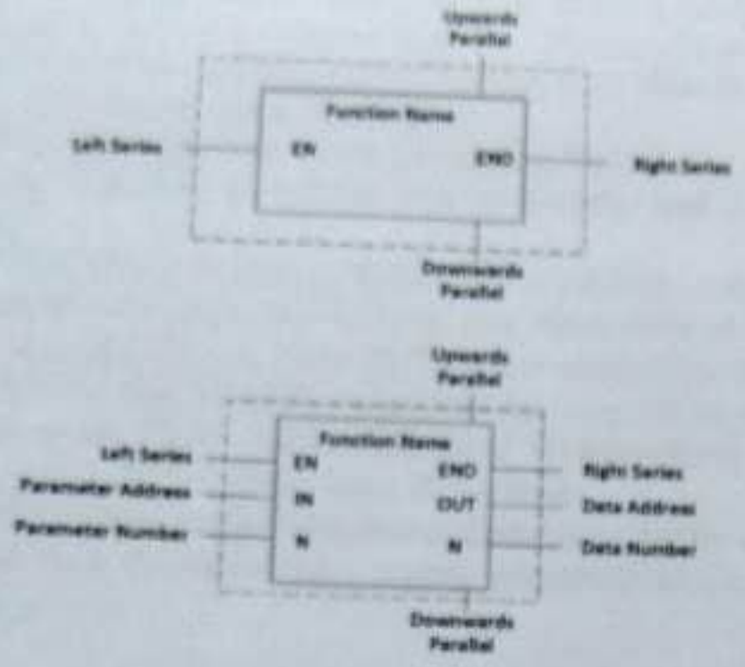

Fig. 2. Two typical design: single component and component with input and output.

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, downwards parallel connection and some auxiliary interfaces.

PT is an attribute collection of the component, including position, size, comment, etc.

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

FIG. 2 illustrates two component design: single component and component with input and output. The single component has function name, left series, right series, upwards parallel and downwards parallel. In component with input and output, it contains function name, left series, right series, upwards parallel, downwards parallel, parameter address, parameter number, data address and data number.

As showed 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 multi-language 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 PLC in memory is made up of bit data area ($Marea$) and byte date area ($Darea$). Two subset attributes are defined as below.

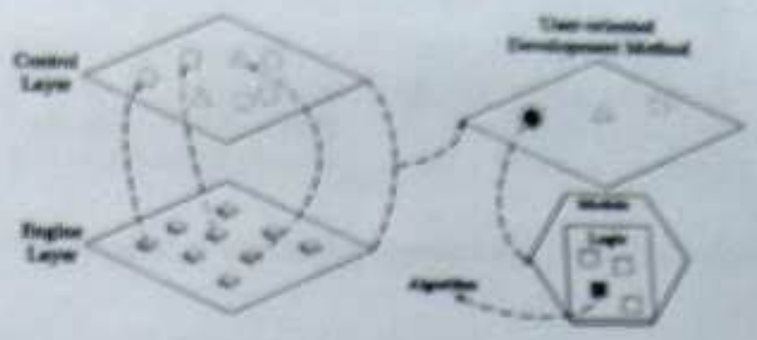

Fig. 3. From the user's point of view, after inducing the multi-language component, the algorithm and logic program could be developed in a uniform PLC platform.

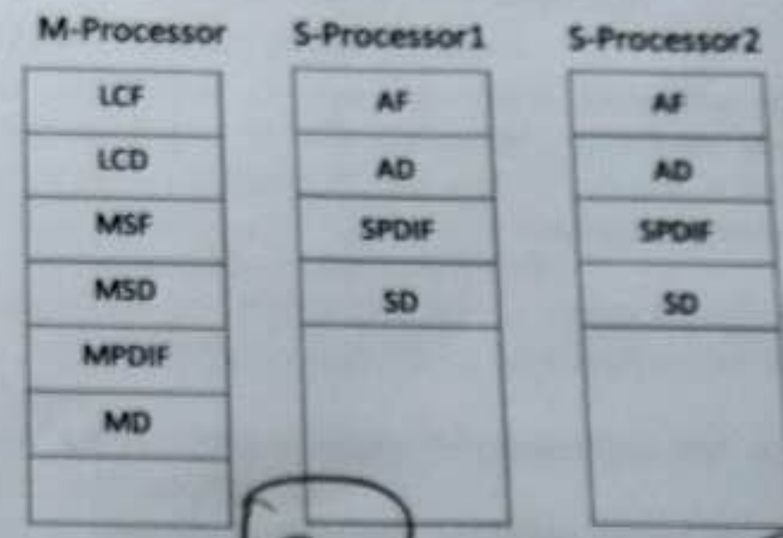

Fig. 4. Memory allocation in master and slave processors $ePLC$.

Definition 1 Set S has B attribute: $\exists(S = \{s_1, s_2, ..., s_i, ...s_n\}) \subset M$ and $(\forall s_i \in S) \in \{\mathcal{F}(s_i) = 0, \mathcal{O}(s_i) = 1\}$

Definition 2 Set S has D attribute: $\exists(S = \{s_1, s_2, ..., s_i, ...s_n\}) \subset D$ and $\forall s_i \in S$ has 4 Bytes

Where $\mathcal{F}(s_i)$ represents the value of s_i is 0, $\mathcal{O}(s_i)$ represents the value of s_i is 1.

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

LCF (Logic Control Flag Area): the flag are used to start the modules.

LCD (Logic Control Data Area): these data will be used to delivery to algorithm.

AF (Algorithm Flag Area): it includes algorithm flag of execution(AFE) and algorithm flag of state(AFS).

AD (Algorithm Data Area): these data help specified algorithm executing.

MSF (Message Flag Area): it includes system message flag (SMF) and user customized message flag (UMF). Both of them have B attribute. SMF is the neccesary message for system execution including start module flag, alarm flag, etc. UMF could be defined by users.

MSD (Message Data Area): it is used to transfer message information which includes system message data area (SMD) and usr message data area UMD.

$MPDIF$ (Master Processor Data Interaction Flag Area): it contains start data transfer flag from master to slave (MS),

occurs when using $\mathcal{P}_{mts}$ and $\mathcal{P}_{mts}$. Basic execution units of control thread are shown as follows:

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

Motion thread contains the following basic execution units:

M_1 start the algorithm.
M_2 execute the algorithm.
M_3 feedback the data to control thread by $\mathcal{P}_{mts}$.
M_4 End algorithm.

Two cases shown in FIG.8 are explained as below:

Case one: execution of control thread and two algorithm threads among three processors. The CT(Control Thread) traverses LCF, finds md_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 SD of processor 1. AT(Algorithm Thread) 1 executes M_1 unit, executes C_2 after transferring data from SD to AD, runs M_3 unit, feedbacks data to CT. When the ap_j finishes, AT execute M_4 and informs CT 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 SD of processor 2, AT 2 executes M_1 unit, executes M_2 after transferring data from SD to AD. When the ap_{j+1} finishes, AT 2 executes M_4 unit and informs CT the end of ap_{j+1}. CT executes C_3 to finish the process.

Case two: execution of control thread and two algorithm threads among three processors together with message mechanism. The CT(Control Thread) traverses LCF, finds md_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 SD of processor 1. AT(Algorithm Thread) 1 executes M_1 unit, executes C_2 after transferring data from SD to AD. During the execution of md_i, AT executes C_5 to broadcast the message smf_x to inform md_{j+1} to run. After CT executes C_5, md_{j+1} gets data and start and then CT finds ap_{j+1} in md_{j+1}, executes C_1 and C_2, transfers the data from LCD to SD of processor 2, AT 2 executes M_1 unit, executes M_2 after transferring data from SD 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, then CT executes C_3 to finish the process.

D. State Transition Mechanism

The state collection of master processor, $MPS = \{mstop, mrun\}$, where $mstop$ means stop state and $mrun$ means run state. The state collection of slave processors, $SPS = \bigcap_{i=1}^{n} SPS_i$, where SPS_i is the state collection of the ith processor and $SPS_i = \{sstop_i, srun_i, sready_i\}$, $sstop_i$ is the stop state, $srun_i$ is the run state and $sready_i$ is the ready state. The state transition mechanism of master processor and slave processors is shown in FIG. 9 and described as below.

$$\begin{cases} mstop \rightarrow mrun \Leftarrow \exists lcf_i \in LCF = 1 \\ mrun \rightarrow mstop \Leftarrow \forall lcf_i \in LCF = 0 \\ sstop \rightarrow sready \Leftarrow \exists sa_i \in SA = 1 \\ sready \rightarrow srun \Leftarrow \exists afe_i \in AFE = 1 \\ srun \rightarrow sstop \Leftarrow \forall afe_i \in AFE = 0 \end{cases} \quad (6)$$

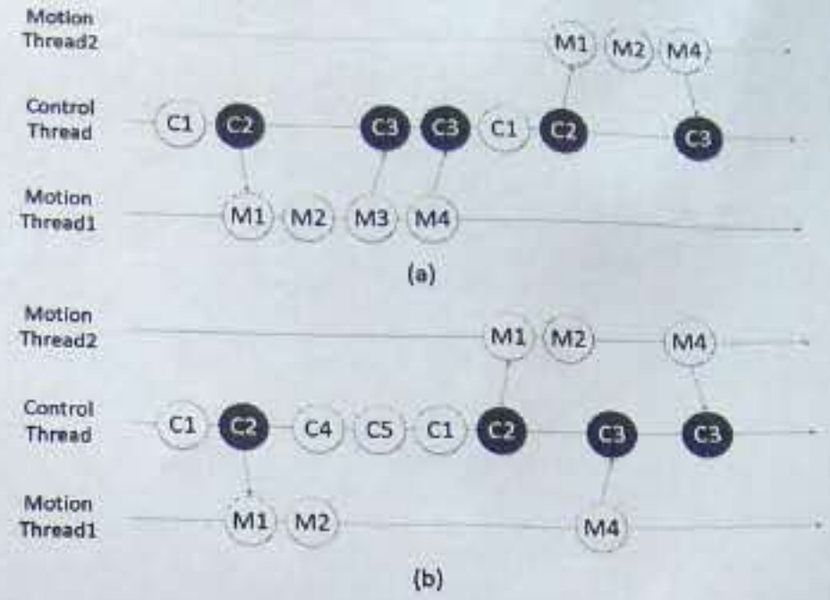

Fig. 8. Execution of control thread and two algorithm threads among three processors with and without message.

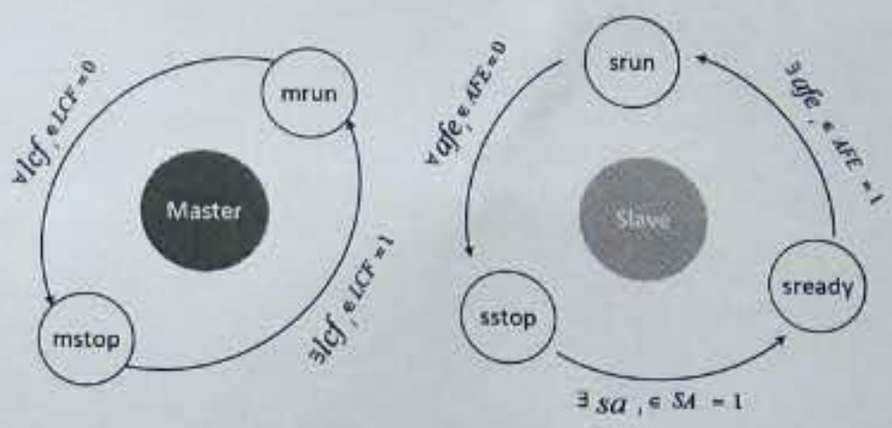

Fig. 9. State Transition Mechanism.

IV. Experiment

A. Distributed Control System

As show in FIG. 10, we verify the proposal development method on a 200 t IMM. The TI F28M35 chip is chosen as the main chip of ePLC. It has two cores: a TI C28x and an ARM Cortex M3. Considering the DSP is more suitable for motion control, Cortex M3 is chosen as master processor and C28x is chosen as slave processor. The ePLC has a RJ45, a RS232 and a CAN. RJ45 is used to download program, RS232 is for connecting with HMI and CAN is designed to extend the DI\O and AI\O. HMI could be customized by users.

B. Software Structure

FIG. 11 shows uniform development platform developed by ourselves. In the dotted line box of FIG. 11 (a), it is the C language component and FIG. 11 (b) is its partial code. This component represents to output a small velocity and pressure in setup mode.

After the design of every used components, we design the modules according to Table I. Additionally, a special module is designed to control the execution flow of all modules with the SMF and SMD.

Three typical requirements using the proposed user-oriented development method are discussed below: