Heterogeneous Multi-platform Code Generation Based on IMCL Model

Ju Li†, Jianqi Shi†, Yanhong Huang†*, Jiangtao Wang†
† National Trusted Embedded Software Engineering Technology Research Center
East China Normal University, Shanghai, China
Email: {jli, jqshi, jtwang, yhhuang}@sei.ecnu.edu.cn

***Abstract*—Model-driven code generation technology has been widely used in system design. Most of the current mainstream model development is code generation for a single platform. However, for heterogeneous systems, due to its multi-platform nature, it is often difficult to design a system using a single model-driven approach. Therefore, we propose a code generation approach that can be used from a single model language to a variety of different target platforms. By analyzing the syntax tree of the IMCL model, a variety of platform codes, including FPGAs, PLCs, and PCs, are generated under given target language conversion rules. Our research can improve the flexibility and practicality of model development. It can help developers focus on the logical design of the system and improve the efficiency of developers.**

I. Introduction

Model-driven architecture (MDA) was proposed by the Object Management Group (OMG) in 2001[1]. It is a software design approach which defines system functionality using a platform-independent model by an appropriate domain-specific language (DSL). By the abstraction of the DSL model, the system architecture can be represented by less code in DSL than other programming languages (Java, C++, C#). Furthermore, programs written in domain-specific languages are easier to understand and even some of the DSLs are graphical languages, which can be used to eliminate the gap between the business logic and technology implementation. Since the model-driven architecture is platform-independent, many interpreters are developed to translate DSLs to platform-dependent languages, such as Simulink and MyGenerator. Also, many developing frames for automatic code generation are proposed, such as Acceleo, which is developed by OMG. With the development of the automatic code generation techniques, changes in system design will not lead to the irreparable impact on the development cycle. By using MDA in our developing process, we can not only perform the syntax checks or static code analysis on the program but also verify the functional properties of the program on the abstractive level, so as to reduce the errors in the implementation.

Due to the fact that there are many heterogeneous devices in complex industrial control systems and their programs run on FPGAs[2], PLCs[3], or PCs, the testing of such industrial control systems faces lots of challenges. Since the MDA is platform-independent, we can use it to solve this testing problem. Therefore we proposed the modeling language IMCL in our previous work[4]. This event-triggered language can describe the physical resources and the system in one unified model. And some reliable and efficient decomposition and collaboration algorithms can be performed based on IMCL models, which means the sub-models after decomposition can be used to represent the programs running on different platforms. Since the sub-models which represent programs running on different platforms are described in the abstract modeling language IMCL, we need to supply some missing details about the software and hardware configurations according to the different platforms before presenting a runnable solution on the specific platform.

In this paper, we propose resource layer to map the computing unit with the control computing capability to the physical resources in the system. Combining with this configuration, we present the formal model of our IMCL models and design some platform-specific rules to convert the sub-models into three kinds of languages (VHDL[5], 61131-3[6], and C). With the help of IMCL, we can design the whole system in a unified form, decompose the system model consistently and generate multi-platform code automatically.

Outline

he remainder of this paper is organized as follows. In Section 2, we introduced IMCL model of complex systems and the formal definition of the model. Section 3 present the approach that how to generate object code from the population IMCL model, including FPGA, PLC and PC. Base on the IMCL mode, Section 4 describes the conversion rules for converting IMCL models into target-platform programs from the perspective of code generation. Section 5 shows the case study of one actual example. Section 6 is the conclusion and our work.

Related Work

Nowadays Model Driven Development (MDD) is becoming more and more popular in the development of industrial control systems. [7] proposed a model-driven IEC 61131-based development process. It discussed three notations (IEC 61499, UML and SysML) as candidates to model the industrial applications. Among the three languages, SysML can be used to define a completely graphical environment that will allow the industrial automation developer to create the model of

*Corresponding Author

the application and automatically transform it to IEC 61131 specification. Formal languages can be used for verification of systems and are therefore particularly popular in model-driven development. As a formal method for system-level modeling and analysis, [8] chose Event-B to reconstruct a subset of the original specification of the CDIS system. This reconstruction overcame three key difficulties of the original formalization, namely the difficulty of comprehending the original specification, the lack of any mechanical proof of the consistency of the specification and the difficulty of dealing with distribution and atomicity refinement.

As FPGA is used in many industrial devices, some automatic code generation researches about VHDL have been done by the researchers. [9] presented the code generation part of ROCCC, and open framework built on the SUIF platform that compiles C programs to VHDL. It describes a novel and improved implementation of the smart buffer that (1) supports loops having multiple input and output arrays and (2) is more area and clock-cycle efficient. Compared with [9], [10] generated VHDL code from a more abstract modeling language UML. Currently, there are some works and commercial tools to generate source code from UML specifications to mainstream languages, such as C++ and Java. However, there are few works addressing the automatic source code generation for VHDL language. This work has developed a first set of mapping rules to generate VHDL from UML models. Besides, IEC 61131-3 language is widely used on the PLCs. Hence there are some techniques about code generation of IEC 61131-3 language. In [11], an approach was developed, which allows to generate IEC 61131-3 code from an UML-model and to import it into soft-PLCs automatically. According to the approach, a prototype was used to demonstrate that automatic code generation for automation technology can be achieved through pragmatic application of UML. An interesting work has been developed in [12], which describes a control logic implementation approach based on discrete event models in form of finite state machines and Petri nets. In this approach, a transformation of the models into an IEC 61131-3 compliant code that can be translated and download into a standard industrial PLC has been proposed.

Most of the above MDD techniques can automatically generate code for a single platform and our research in this paper is to generate code for a variety of different system platforms.

II. Preliminary

IMCL is an event-triggered language. The purpose of IMCL modeling language design is to model the actual industrial control domain system.

A. IMCL Mdodel

The modeling of complex industrial systems has different angles. The IMCL modeling method is based on the idea of refinement: modeling system functions, control logic, system resources, etc., layer by layer.

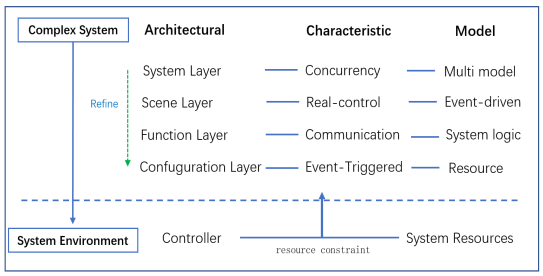

Fig. 1. IMCL Model

- **System Layer** The system layer embodies the intuitive composition of a model. Usually, the system is composed of modular or functional components. For heterogeneous systems in the field of industrial control, controllers or processors with computational control capabilities within them can all run independently of each other. From the overall behavior of the system, the operation process is highly concurrent.
- **Scene Layer** The scenario layer describes the logical relationships between the independent components in the system, namely the control flow and interaction rules. All scenarios include system-specific task execution sequence, event triggering, message transmission, and so on. The running process of a system can be seen as the change of the scene, and can also be seen as the interaction process within the system. In the IMCL modeling process, the scene layer corresponds to events in the system.
- **Function Layer** The functional layer describes the system's behavioral process. Relative to the scene layer. The functional layer can be seen as the refinement of the scene layer. It describes the details of the implementation of the scene, including requests for various types of messages that occur in the scene, data calculations and interactions, and the scheduling relationships between the controller and the device.
- **Configuration Layer** The configuration layer reflects the mapping relationship between the computing unit with the control computing capability and the system physical resources in the system. Each computing unit can only control and schedule specific physical resources. The actual description of this constraint relationship is the respective functions of different processing units. By configuring this type of resource constraint relationship, the internal structure of a complex system can be more realistically reflected.

The entire system modeling process mainly includes the following three aspects:

1) **Unified definition of resources** Representations of physical resources are different based on diverse industrial environment, for instance, sensors, read-write devices, and the other resources. Considering their effects on the whole system, we describe all those resources as

variables to unifying definition of resources.

2) **Modelling the system** Observing its behavior, the nature of the system is gathering functions, read-write operations, and other actions together. Similar to physical resources, we model them as execution expressions. Multiple execution expressions in one specific order can make up one trigger event
3) **Resource Constraint** It describes the constraints that physical resources are limited available for specific controllers.

B. IMCL formal model

In order to better understand the characteristics of the IMCL model, we introduce the characteristics of the model from a formal point of view. For any system, modeling from the perspective of system trigger events using IMCL language, the system model can be expressed as follows:

$$Prog = \underset{i=1}{\overset{n}{\bowtie}} T_i, \quad n \in N^+$$

From the model refinement point of view, each system $Prog$ program can be seen as a set of trigger events, and then each trigger event T_i is a set of ordered command expressions, each command expression can be seen as a system execution The smallest unit of calculation. For a system, the IMCL language is used for modeling from the perspective of system trigger events. The system model can be expressed as follows:

Definition 1. ***IMCL Model*** $IMCL = \langle V,\ T^*,\ R^*,\ C^* \rangle$

The IMCL model is characterized by event triggering and seen as a combination of events. The system variable V is a set of different variables; T^* is the event set, all events are distributed concurrently; R^* represents the resource definition of the program, which is regarded as a special variable in the model; C^* represents the resources constraint in the system relationship.

Definition 2. ***Variable*** $V = V_{in} \cup V_{out} \cup V_{mess} \cup V_{local} \cup V_{res}$

The set of variables represents how the variable data is expressed in the system. V can be divided into five kinds of sets: the input variable set is represented as V_{in}, the output variable is V_{out}, the local signal variable is V_{mess}, the local variable set is V_{local}, and the physical resource description set V_{res}. These variables run through all state transitions during program execution and involve various types of operational rules for the system.

Definition 3. ***Trigger Event*** $T = \langle id,\ c,\ E^*,\ V_t,\ R_t \rangle$

id represents the unique identifier of the event, each event can represent T_{id}; c is the event trigger condition and it can be a specific conditional expression or a bool value. It is a prerequisite for event execution migration. E represents the set of tasks contained in the event; $V_t = \{Vglobal; Vlocalg\}$ represents the set of variables contained in this event, where V_{global} is the program global variable and V_{local} is the set of local variables inside the event T. $R_t \subseteq R^*$ indicates the physical resources that event mapped. The conditional triggering relationship of an event can be represented by the following expression:

$$(1)\langle id, c, E^*, V_t, R_t \rangle \rightarrow false \Rightarrow \langle id, c^{'}, E^*, V_t^{'}, R_t \rangle$$
$$(2)\langle id, c, E^*, V_t, R_t \rangle \rightarrow true \Rightarrow \langle id, true, E^{*'}, V_t^{'}, R_t \rangle$$

Definition 4. ***Task*** $E = E_{com}|E_{inv}|E_{branch}|E_{loop}|E_{ch}|E_{sh}$

The task represents the unit of program execution. Where E_{com} represents a general assignment operation or numerical operation. E_{inv} indicates that the task is to interact with system physical resources; E_{branch} indicates conditional branch; E_{loop} indicates program loop execution; E_{ch} indicates execution statement of system communication process.

Definition 5. ***Branch*** $E_{branch} = \langle b,\ E_{if},\ E_{else} \rangle$

The branch is the conditional statement, which means that the program enters the selected state. We define b as the branch condition. E_{if} will be chosen if b is true, otherwise the E_{else}. We define $e_0 \in E_{if}$ and $e_1 \in E_{else}$, $\triangle$ as the program execution current state, and $\triangle^{'}$ as the program change state. The program's transition relationship can be expressed in the following form:

$$(1)\ \langle b, \triangle \rangle \rightarrow true, \langle e_0, \triangle \rangle \rightarrow \triangle^{'}$$
$$\Rightarrow \langle if\ b\ then\ e_0\ else\ e_1, \triangle^{'} \rangle \rightarrow \langle e_0, \triangle^{'} \rangle$$
$$(2)\ \langle b, \triangle \rangle \rightarrow false, \langle e_1, \triangle \rangle \rightarrow \triangle^{'}$$
$$\Rightarrow \langle if\ b\ then\ e_0\ else\ e_1, \triangle^{'} \rangle \rightarrow \langle e_1, \triangle^{'} \rangle$$

Definition 6. ***Loop*** $E_{loop} = \langle cond, E_{while} \rangle$

Since the loop statement has been used to indicate that the current program enters a loop execution state, we define b as a loop condition, $\triangle$ as the program execution current state, and $\triangle^{'}$ as the program change state. The execution of the program is as follows:

$$(1)\ \langle b, \triangle \rangle \rightarrow false \Rightarrow \langle while\ b\ do\ e, \rangle \rightarrow \triangle$$
$$(2)\ \langle b, \triangle \rangle \rightarrow true, \langle c, \triangle \rangle \rightarrow \triangle^{''}, \langle while\ b\ do\ c, \triangle^{''} \rangle \rightarrow \triangle^{'}$$
$$\Rightarrow \langle while\ b\ do\ c, \triangle \rangle \rightarrow \triangle^{'}$$

Definition 7. ***Communication*** $E_{ch} = ch!V_{mess}|ch?V_{mess}$

Where $ch!V_{mess}$ that send messages, is the active process of the program. $ch?Vmess$ that the system receives the message, is a similar to the ready-passive process. $\triangle$ is the current state of the program execution, $\triangle^{'}$ is the state of the program change. We represent the transmission mechanism of the communication.

$ch!Vmess$ defines the process as when the model is actively sending out messages, it will change the system message variables, modify the message variables, and the system state will change. The expression is as follows:

$$\langle ch!, V_{mess}, \triangle \rangle \rightarrow \langle ch!, V_{mess}^{'}, \triangle^{'} \rangle$$

$ch?V_{mess}?$ defines the process as the model is in the process of accepting the message, and its program state remains unchanged until it receives the target message from the channel.

(1) $\langle ch!, b, V_{mess}, \triangle\rangle \rightarrow false \Rightarrow \langle ch!, b, V_{mess}^{'}, \triangle\rangle \rightarrow \triangle$

(2) $\langle ch?, b, V_{mess}, \triangle\rangle \rightarrow true, \langle c, \triangle\rangle \rightarrow \triangle^{''}, \langle ch?, b, V_{mess}, \triangle^{''}\rangle \rightarrow \triangle^{'} \Rightarrow \langle ch?, b, V_{mess}, \triangle\rangle \rightarrow \triangle^{'}$

Definition 8. ***Schedule*** $E_{sh} = \langle a_{data}, \lambda, Dev\rangle$

The resource scheduling E_sh reflects the scheduling relationship between the controller and the physical resources. There are two types of λ, $a_{data} \ll Dev$ and $a_{data} \gg Dev$. $a_{data} \ll Dev$ indicates that the controller schedules acquisition of data to the physical device; $a_{data} \gg Dev$ indicates that the controller schedules transmission of data to the physical device. Since the purpose of the IMCL model is to study the logical function of the system, we use the two forms of to describe the scheduling function between the controller and the physical device.

C. Group model generation

The specific implementation details of the population model generation technique have been proposed in our previous published papers. The advantage of using the ICML modeling method is that we can intelligently split a complex model into multiple sub-models given the constraints of the resource and controller constraints. The sub-models can communicate with each other and achieve the same function as the original model. The generated sub-models correspond to specific target platforms respectively, and the main research work in this paper is to realize the code generation work from the IMCL model to different target platforms.

III. Approach

In the previous section we introduced IMCL model approach to complex systems and the formal definition of the model. Based on this, we will introduce how to generate object code from the population IMCL model.

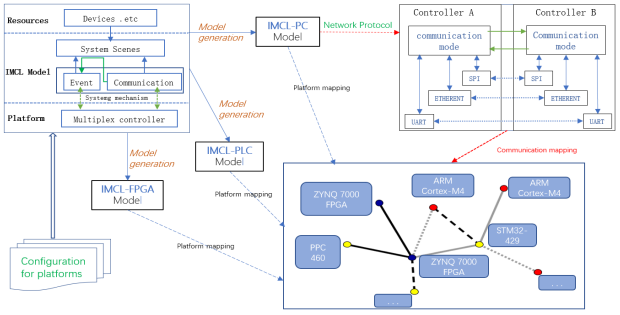

Fig. 2. The approach of IMCL Model to multi-platform code

The specific implementation details of the population model generation technique have been proposed in our previous published papers. The advantage of using the ICML modeling method is that we can intelligently split a complex model into multiple sub-models given the constraints of the resource and controller constraints. The sub-models can communicate with each other and achieve the same function as the original model. The generated sub-models correspond to specific target platforms respectively, and the main research work in this paper is to realize the code generation work from the IMCL model to different target platforms.

A. Conversion of IMCL Model and Heterogeneous Platform

We conduct research on different platforms, including FPGA, PLC, and PC. The research mainly includes how to use IMCL to represent these heterogeneous systems.

1) ***Conversion of FPGA and IMCL Model****:* FPGA (Field-Programmable Gate Array, Field Programmable Gate Array), due to its customizable features, is widely used in medical equipment, rail traffic control and other fields. The system developed by the VHDL language used by the FPGA includes the following basic parts:

- **Library** Declares the repository that the program needs to use, including the std, work, and user-defined libraries. Library contains a variety of design elements, from a program perspective, can be seen as a collection of data.
- **Use** This section is related to the Library and declares the specific resources used by the corresponding resource library in the Library.
- **Entity** This area is the entity declaration of the VHDL program and mainly describes the relationship between the input, output, and ports of the system circuit.
- **Architecture** The architecture is the behavior part of the circuit. The architecture supports the parallel and serial of the program. The main description is its internal implementation process, including data flow, structure description, behavior description and so on.
- **Configuration** The main purpose of the configuration is to select the required units from the library and form the required system. From the perspective of system resources, it is a process of choice and combination.

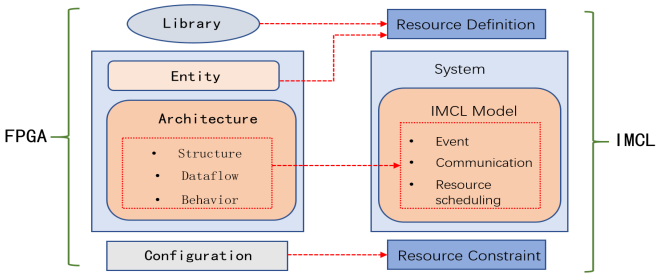

Fig. 3. The transformation architecture of FPGA and IMCL-Model.

Combined with the characteristics of the previously analyzed IMCL model, we can see that there are commonalities between the two architectures. After ignoring the irrelevant platform details, IMCL can model the behavior described by VHDL. As shown in the figure, the library and package structure represented in VHDL can be abstracted into resources in the process of modeling with IMCL after ignoring platform dependencies. The entity represents the design of the circuit

structure of the system and can also be used as a description of resources. The architecture in VHDL mainly includes the structure description, data flow description, and system line description. Essentially, they describe the functional characteristics of the internal structure and correspond to the events described in IMCL.

*2) **Conversion of PLC and IMCL Model**:* PLC (Programmable Logic Controller) is a kind of programmable control industrial control computer. PLC takes the microprocessor as the core and realizes the control of the system through software. There are many types of PLCs, but their structural principles are basically the same: they include processors, storage, I/O ports, and network communications. Take the language IEC 61131-3 language by PLC as an example, the design language includes five forms: LD(Ladder Diagram),IL(Instruction List),FBD(Function Block Diagram),SFC(Sequential function chart),ST(Structured Text).

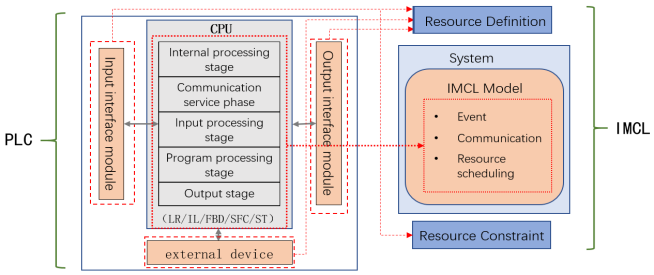

Fig. 4. The transformation architecture of PLC and IMCL-Model.

We can use IMCL to represent the operating mode of the PLC. The working phase of the PLC is to periodically scan cyclically, and it will continue to work when there is no interruption or other situations. The general PLC operating mode can be divided into five phases:

- **Internal processing stage**: detecting the system's current ready state and resetting the internal timer;
- **Communication service phase**: The PLC has a communication function. The external control module can communicate with each other and can also receive signal commands from other controllers.
- **Input processing stage**: read the information data of the mounted peripherals and sample the data into the system at one time.
- **Program processing stage**: This stage is the core stage of the PLC control process and is the main body of the PLC program, including condition control, numerical calculation, and logic conversion. This stage reflects the functional behavior of the system.
- **Output stage**: After the main program runs, data is loaded to the outside through the output mechanism.

A unified description of the external physical resources associated with the input and output modules, peripherals, etc. The abstract resource object is a program variable, which can facilitate resource scheduling and set constraint conditions. For the main program of the PLC, we extract the main part of the communication services, program execution process, and then use an event-driven way to describe. Finally, the PLC application can be described by the IMCL model.

3) Conversion of PC and IMCL Model: PC (personal computer) is often widely applied to industrial systems because of its advantages such as high-speed processing speed, reliable operation platform, mass storage, networking, and friendly human-computer interaction. For example PCBCS, PC can communicate collaborate with other mainstream PC or PLC systems implement complex functional requirements. In PCBCS, PC's communication technology is one of its greatest advantages. PC can be compatible with almost all communication protocols in the mainstream, so it is very beneficial to the design of complex systems. The common PC system design language is C. Due to its good compatibility, portability, and high execution efficiency, it is widely used in industrial-grade system design.

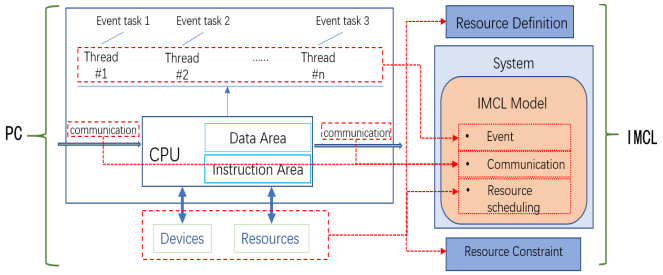

Fig. 5. The transformation architecture of PC and IMCL-Model.

As shown in the figure, a typical PC-style control system design can be represented as shown in the above figure. The CPU is responsible for the execution of the program. The entire system is composed of multiple independent threads. Each thread represents the relevant task. . The system has independent communication, including data input and output. When we use IMCL to model, we represent the multi-threading as a set of concurrent trigger events; the communication of system functions can be represented using the IMCL abstract communication protocol; the control relationship between the system and external device resources, using the resources in IMCL Scheduling to model.

B. Code generation configuration

The essence of the system model is to abstract away some irrelevant details and only pay attention to a research method to study the characteristics of the object. Therefore, when we want to be able to generate code from model automation, we need to supplement the missing details. In the code generation process from IMCL to a specific target platform, configuration information needed includes variable conversion, communication protocol method between systems, and the driving relationship between a controller and specific devices. Here we use *Conf* to represent these configuration: $Conf = \langle V_{map},\ C_{map},\ D_{map} \rangle$.

V_{map} represents the variable mapping relationship between the model and the specified platform controller: $V_{map} = V_{imcl} \rightarrow (V_{plc}|V_{fpga}|V_{pc})$. The V_{map} refers to the variables

$V_{in} \cup V_{out} \cup V_{mess} \cup V_{local} \cup V_{res}$ in IMCL; $V_{plc}|V_{fpga}|V_{pc}$ corresponds to a collection of variables for specific heterogeneous platforms.

C_{map} represents the mapping relationship between the communication method in the model and the communication protocol used by a particular platform: $C_{map} = C_{imcl} \rightarrow (C_{plc}|C_{fpga}|C_{pc})$. $C_{imcl} = ch!Vmess|ch?V_{mess}$ refers to the formal representation of communications in IMCL; $C_{plc}|C_{fpga}|C_{pc}$ refers to the definition and implementation of specific communication protocols for different platforms.

D_{map} represents the mapping relationship between the driver representation in the model and the drivers of controller and devices in particular platforms: $D_{map} = D_{imcl} \rightarrow (D_{plc}|D_{fpga}|D_{soc})$. $D_{imcl} = a_{data} \ll Dev|a_{data} \gg Dev$ refers to the scheduling relationship between the controller and peripheral physical devices in IMCL, and $(D_{plc}|D_{fpga}|D_{pc})$ corresponds to specific target platforms that need to implement the device scheduling driver.

IV. Rules

The IMCL model has features such as event triggering, message communication, and resource scheduling. This section describes the conversion rules for converting IMCL models into target-platform programs from the perspective of code generation. Common techniques for code generation are based on ASTs, and so are ours. The abstract syntax tree is also called the AST syntax tree, which is the tree structure corresponding to the source code syntax. That is, for source code in a specific programming language, statements in the source code are mapped to each node in the tree by constructing a syntax tree. In the tree structure on the basis of IMCL, given by the code generation rules, the model may be implemented to generate the code for the target platform.

As shown in 1, the first step in heterogeneous multi-platform code generation is that we need to convert the IMCL code into a structured syntax tree AST. Next, we use the ANTLR tool to generate an abstract syntax tree of the IMCL model under the IMCL syntax. Then through the depth traversal of the tree structure, analyze the type of each node. Finally, we use the corresponding rules to convert according to the type of each node.

Next we will introduce the specific details of these different rule conversions:

*a) **Rule 1: Conversion of variables**:*

There are 5 forms of variables of the IMCL program. These variables cover the basic variable types of IEC 61131-3, VHDL, and C. According to the definition of$V_{map} = V_{imcl} \rightarrow (V_{plc}|V_{fpga}|V_{pc})$, all variables in different controller programs can be represented by *global variables*, *local variables*, and *static variables*. For each controller program, it is converted into a specific variable type in the program, such as *int*, *string*, and so on.

*b) **Rule 2: Conversion of events**:*

The form of the IMCL IPC model is a set of events in which events are concurrent. After the trigger condition of any event is satisfied, the subject of the respective event will be executed. Different programming languages have different ways of expressing events, and they will be converted separately for each of the three languages:

- **IEC 61131-3 Language**: there is only SFC that expresses concurrency.

  An SFC program is defined as a tuple $SFC = \langle V, S^*, s_0, L^*, pDv^* \rangle$, where V represents the set of variables, S^* represents the set of steps, s_0 represents the initial step of the SFC, and L^* represents the interior The set of instruction programs, pDv^* represents a collection of concurrent branch structures.

  As described above, from an event-driven perspective, all ICML events can be considered as branches of the SFC program. That is, each branch of pDV^* can be considered as an event in IMCL. So we use SFC as an architectural language to describe the structure of the entire system. In the body of the specific trigger event in the IMCL, we use the ST language in 61131-3 to describe the details of the event logic equivalently. For detailed rules, see Rule 3, Rule 4, and Rule 5.
- **VHDL Language**: In VHDL, programs are represented in the form of architecture, and multiple events in a program can be represented using the process structure. Because in the architecture, each process is a program block, and all program blocks are parallel, it can equivalently represent each event with each process.

  A process in an architecture can be regarded as a tuple $PROCESS = \langle name, V, cond^*, L^* \rangle$, where $name$ represents the process name and a representation of a process structure, then V represents a collection of variables of the process, $cond^*$ indicates a trigger condition, and L^* indicates a set of programs that are executed internally.

Algorithm 1: Collaboration of models.

Input: (1) IMCL code; (2) Rules of specific target platform;

Output: Target platform code;

```
1  AST_Imcl ↔_AST IMCL ;
2  function DeepFirstVisit(AST_Imcl)
3      for ∀ node ∈ AST_Imcl do
4          type = getRuleType(node) ;
5          if type ∈ Rule1 then
6              variableHandler();
7          end
8          if type ∈ Rule2 then
9              eventHandler();
10         end
11         if type ∈ Rule3 then
12             structHandler();
13         end
14         if type ∈ Rule4 then
15             communicationHandler();
16         end
17         if type ∈ Rule5 then
18             scheduleHandler();
19         end
20     end
```

Since VHDL program architecture can be viewed as a collection of process. Each T_i in $\overset{n}{\underset{i=1}{\bowtie}} T_i$ corresponds to a *PROCESS*.

- **C Language**: In C language, in order to represent multiple events and the concurrent relationship between events, we use *thread* to represent the event.

  All events are treated as a tuple $Thread = \langle id, V^*, L^* \rangle$, where id indicates that the process is uniquely identified, V^* indicates the data space of the thread, and L^* identifies the thread's execution body.

  In C language, the main body of the program can be seen as multiple parallel threads, each thread can be seen as an event task. Each T_i in $\overset{n}{\underset{i=1}{\bowtie}} T_i$ is a Thread. A multi-event IMCL model can then be converted into a multi-threaded C program.

*c) **Rule 3: Conversion of structured statement**:*

IMCL contains the conditional statement **if**. A Case statement in IMCL is defined as a tuple $\langle caseExp, caseBody^* \rangle$ where *caseExp* represents the expression of the case statement, *caseBody* is defined as $\langle caseValue; L^* \rangle$, and *caseValue* represents the value of the expression into the case clause, L^* denotes List of statements executed by each case clause.

IMCL contains loop statements **while**. A loop statement of IMCL is defined as a tuple $\langle whileExp; L^* \rangle$, where *whileExp* represents the loop condition in the while statement L^* represents a list of statements executed in each loop.

Conditional statements and loop statements are the basic structures in the programming language. The semantics of the conditional structure and loop structure of all languages are equivalent. So IMCL and IEC 61131-3, VHDL, and C have common features in the *if* and *while* structures, and the conversion between them is direct and equivalent.

*d) **Rule 4: Conversion of communication**:*

The communication method in the system model is abstract. It only contains the communication method and communication content, but it does not care about the specific implementation details of the communication. The communication mode of IMCL abstract communication, such as protocols such as UART, ENHENT, SPI, does not pay attention to how the bottom layer is implemented. The bottom layer includes different links for communication, such as network ports and serial ports. All models can send content through the communication module to the channel through message binding. The subsystem that needs to receive the message will automatically obtain the data in the bound channel. Therefore, when researching code generation, communication functions need to be implemented for specific communication protocols between devices. In our research, we encapsulate the mainstream centralized communication protocol into a function interface. When users generate code, they only need to select the corresponding interface.

According to the previously defined configuration $C_{map} = C_{imcl} \rightarrow (C_{plc}|C_{fpga}|C_{pc})$, we defined the interface under the communication rules as follows:

	PLC(61131-3)	**FPGA(VHDL)**	**PC(C)**
UART	ST_UART	Entity_UART	C_UART
Entherent	ST_Entherent	Entity_Entherent	C_Entherent
SPI	ST_SPI	Entity_SPI	C_SPT
...	...	...	...

In particular, since the ways in which the various controllers execute the programs are different, the expressiveness and manner of the program languages must be taken into account when converting between languages. Below we describe how the three languages convert IMCL message.

- **IEC 61131-3 Language**: The execution mode of the program main body cycle is $input \rightarrow processing \rightarrow output$, and the atomicity is continuous without interruption. A segment of the IMCL program is continuous and may contain input and output of messages in the intermediate process. Therefore, in order to convert the specific process of events in IMCL to ST language in IEC 61131-3, we use label to convert. The specific principle is: the main body of the IMCL event is divided into a plurality of phases with the communication statement as a mark, so that the ST program can be completed using several execution cycles.
- **VHDL Language**: For all communication modules, we can use Entity. Regardless of UART or Entherent, we use the corresponding entity to achieve. For a specific communication protocol, we implement it as two entities: an accepting entity and a sending entity. For example, the Entity_UART in the table indicates that for the UART communication mode, the specific communication process is implemented by one entity.
- **C Language**: For the code generation of the system to communicate with an external controller, we use an abstract interface to represent it. For different communication methods of different protocols, we divide the message into two parts: accept and send, and pre-implement their functions in the defined library. In the code generation, we call the specific communication function in the predefined library for code generation.

*e) **Rule 5: Conversion of schedule**:*

Physical resources of the physical device industrial control system environment is diverse, including the number of diversity, functional diversity. For the relation $D_{map} = D_{imcl} \rightarrow (D_{plc}|D_{fpga}|D_{soc})$, all device scheduling controls in the IMCL model use a formal representation, but when studying how to switch to a specific target device, it needs to be based on the actual controller and hardware Scheduling protocol to implement the scheduling function.

V. Case Study

An example will be introduced to show how the heterogeneous multi-platform code generation based on IMCL model.

A. The IMCL group model

The Fig 6 is a computing system composed of three different platforms. Each of the three platforms, FPGA, PLC and

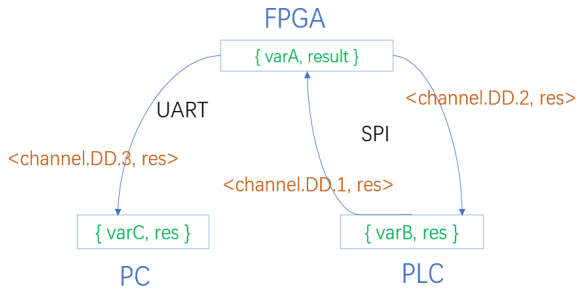

Fig. 6. Heterogeneous computing systems of FPGA, PLC and PC

FPGA Model

```
program:FPGA(VAR: res, varA){
  TRIGGER(TRUE){
    res:=4;
    CHANNEL.DD.!2:res;
    CHANNEL.DD.?1:varA;
    WHILE(varA > 0){
      res:=res+1;
      varA:=varA-1;
    }
    CHANNEL.DD.!3:res;
  }
}
```

PLC Model

```
program:PLC(VAR: res, varB){
  TRIGGER(TRUE){
    varB:=10;
    CHANNEL.DD.?2:res;
    IF(res % 2 == 1){
        varB :=res+varB;
    }
    CHANNEL.DD.!1: varB;
  }
}
```

PC Model

```
program:PC(VAR:res, varC){
  TRIGGER(TRUE){
    CHANNEL.DD.?3:res;
    varC :=res+varC;
    varC>>Screen;
  }
  TRIGGER(TRUE){
    if (varC > 100){
      "Waring" >> Screen;
    }
  }
}
```

Fig. 7. IMCL group models of FPGA, PLC and PC

PC, has its own independent computing and system control functions. Among them use SPI agreement between FPGA and PC, use UART agreement between FPGA and PLC. Each platform's program contains a set of specific variables, and they work together to achieve the ultimate computational task.

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity UART is
    Port (varA : out_uart : out  INTEGER;
          clock : in STD_LOGIC);
end UART;

entity SPI is
    Port (varA : in_spi : out  INTEGER;
          out_spi : out  INTEGER;
          clock : in STD_LOGIC);
end SPI;

architecture Behavioral of FPGA_MAIN is
Variable varA, res: INTEGER;
begin
FPGA: process(clock)
    begin
        res:=4;
        out_uart <= res;    --// sent res
        varA <= in_uart;    --// receive varA
        While varA > 0 Loop
            res:=res+1;
            varA:=varA-1;
        End Loop;
        out_spi <= res;
    end process FPGA;
end Behavioral;
```

61131-3

```
(* Function receive_SPI *)
FUNCTION receive_SPI : REAL
VAR_INPUT
      fromId: REAL;
END_VAR
(* detail of the SPI that receive message*)


(* Function send_SPI *)
FUNCTION send_SPI : REAL
VAR_INPUT
      to: REAL;
END_VAR
(* detail of the SPI that sent message*)


(* Main body of program*)
VAR
      res, varB: REAL;
END_VAR

varB:=10;
res:= receive_SPI(2);
IF(res % 2 = 1) THEN
    varB := res+vaarB;
END_IF
send_SPI(varB)
```

C

```
#include <pthread.h>
...

extern void receive_Uart(string &fromId);
extern void driver_Screen(Data &data);

void trigger1() {
    res = receive_Uart(2);
    varC = res + varc;
    driver_Screen(varC);
}

void trigger2() {
    if (varC > 100)  {
        driver_Screen("Warning");
    }
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, trigger1, NULL);
    pthread_create(&t2, NULL, trigger2, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    // ...
    return 0;
}
```

Fig. 8. Heterogeneous Multi-platform Code Generation based on IMCL group models(VHDL, 61131-3 and C language)

Firstly, we use IMCL to describe the entire system. The system model includes three IMCL models that correspond to FPGAs, PCs, and PLCs. Each model contains control functions such as logic operations and communications. The three models communicate data through communications to perform collaborative calculations. The entire system's IMCL model is shown in Fig 7.

B. Multi-platform Code Generation based

The above three platforms: FPGA, PLC, PC constitute a heterogeneous system. According to the defined five kinds of transformation rules, by analyzing the abstract syntax tree of the IMCL model program, the IMCL group model can finally generate the target platform code corresponding to the corresponding system.

The code in Fig 8 shows the Heterogeneous Multi-platform Code generated by the defined approach. In the VHDL language, we use two Entity to define the communication protocol UART and SPI respectively, and use Behavior to define the function of the main body. In 61131-3, we define the communication function SPI through the functions receive_SPI and send_SPI, and use ST language to describe the system's function. In the C language, we implement the communication function by defining the receive_Uart interface, and define the information scheduling relationship to the device Screen by defining the function driver_Screen. Based on the analysis of the equivalence relationship between the three platforms and IMCL's system descriptions, we can conclude that our approach to approach code generation is feasible and has application value. In future work, we will continue to study more efficient conversion methods, and we will explore the conversion rules between IMCL and more platforms.

VI. Conclusion

Model-driven code generation techniques have been widely used in system design. For heterogeneous systems, due to their multi-platform nature, it is often difficult to design a system using a single model-driven approach. Therefore, we propose a code generation method that can be used from a single model language to a variety of different target platforms. In this paper, we introduce IMCL features and modeling methods. We have given an approach to converting code from IMCL to the corresponding platform. In this article, we introduce the functional equivalence between the IMCL model and FPGA, PLC, and PC systems. Focused, we have given the rules for conversion between languages. On this basis, by analyzing the syntax tree of the IMCL model, under the given target language conversion rules, we can generate the framework code of the three types of platforms. Our research can improve the flexibility and practicality of model development, allowing developers to put time and energy into the logical design of the system and improve the efficiency of developers.

Acknowledgment

The authors would like to thank...
The authors would like to thank...
The authors would like to thank...
The authors would like to thank...
The authors would like to thank...
The authors would like to thank...

REFERENCES

[1] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *International Conference on Object-Oriented Information Systems*. Springer, 2002, pp. 290–297.

[2] E. Monmasson and M. N. Cirstea, "Fpga design methodology for industrial control systemsa review," *IEEE transactions on industrial electronics*, vol. 54, no. 4, pp. 1824–1842, 2007.

[3] S. Abdallah and S. Nijmeh, "Two axes sun tracking system with plc control," *Energy conversion and management*, vol. 45, no. 11-12, pp. 1931–1939, 2004.

[4] J. Li, J. Xiong, X. Mao, J. Shi, X. Ye, and Y. Huang, "Decomposition and collaboration of industrial control system with resource constraints," in *Engineering of Complex Computer Systems (ICECCS), 2017 22nd International Conference on*. IEEE, 2017, pp. 162–165.

[5] P. P. Chu, *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2011.

[6] K. H. John and M. Tiegelkamp, *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.

[7] K. Thramboulidis and G. Frey, "Towards a model-driven iec 61131-based development process in industrial automation," *Journal of Software Engineering and Applications*, vol. 4, no. 04, p. 217, 2011.

[8] A. Rezazadeh, N. Evans, and M. Butler, "Redevelopment of an industrial case study using event-b and rodin," 2007.

[9] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for fpgas," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 1, p. 6, 2008.

[10] T. G. Moreira, M. A. Wehrmeister, C. E. Pereira, J.-F. Petin, and E. Levrat, "Automatic code generation for embedded systems: From uml specifications to vhdl code," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*. IEEE, 2010, pp. 1085–1090.

[11] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a uml model to iec 61131-3 and system configuration tools," in *Control and Automation, 2005. ICCA'05. International Conference on*, vol. 2. IEEE, 2005, pp. 1034–1039.

[12] G. Music, D. Gradisar, and D. Matko, "Iec 61131-3 compliant control code generation from discrete event models," in *Intelligent Control, 2005. Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation*. IEEE, 2005, pp. 346–351.