This is a repository copy of *Improvement of type declaration of the IEC 61499 basic function block for developing applications of cyber-physical system*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/92364/

Version: Accepted Version

Improvement of type declaration of the IEC 61499 basic function block for developing applications of cyber-physical system

Shiyong Wang, Chunhua Zhang* Dongyao Jia

South China University of Technology, Guangzhou, China
University of Leeds, Leeds, UK

Abstract: The cyber-physical system relies on a network of physical devices. The IEC 61499 standard entails a systematic solution to distributed system development. The basic function block (BFB) is the essential construct of the IEC 61499 architecture. However, the execution semantics of the BFB is not well defined by the standard, which leads to a part of the semantic ambiguity. In this paper, we contribute to improve BFB type declaration by proposing a compact interface model and a strict execution control chart (ECC) model. The improved BFB exhibits less semantic ambiguity, is easier to be created, and is more convenient to be applied in a function block network than the standard BFB.

Keywords: BFB, ECC, event type, IEC 61499.

1. Introduction

With the development of microprocessors and network technologies, the embedded devices are enabled with the capabilities of computing, control, and communication. Nowadays, a collection of embedded devices can be interconnected to serve a system-wide purpose. This lays a solid foundation for cyber-physical system (CPS) that addresses the deep integration of virtual computation with physical processes [1]. Although it is believed to see the extensive application of CPS in many domains from industry to agriculture or even city governance and everyday life [2], developing and managing such applications that run over a network of distributed devices is still a challenge. The IEC 61499 standard [3] has been approved to provide a language independent architecture for developing and maintaining the distributed applications, so it is a promising way to support CPS programming [4].

The standard specifies three types of function blocks (FBs), i.e. the Basic Function Block (BFB), the composite function block (CFB), and the service interface function block (SIFB). An Application, i.e. the solution to a measurement or control problem, is programmed in the form of the function block network (FBN) consisting of interconnected FBs. Object-oriented encapsulation of data and functions and the event-triggered mechanism enable the FBs to be reusable, portable, distributable, etc. Thus the FB architecture is considered to be an enabler of distributed and intelligent automation [5].

Semantic ambiguity exists in the FB architecture due to the incomplete definition of the standard [6]. Two aspects of effort were carried out to solve the problem. On one hand, some rules were proposed to set up a determinate execution modal. Vyatkin, et al. [7] proposed six postulates to construct a sequential axiomatic model for BFB execution. Those postulates mainly focused on event related issues. In their further work [8], a set of graph transformation rules were proposed to

rebuild the execution control chart (ECC) of the BFB. The main purpose was to get rid of potential deadlock states by removing arcs without event input variables. As to the FBN, Sünder, et al. [9] studied the execution model of CFB and Subapplication (a type of FBN that can be distributed). They stated that a CFB could be viewed as an entity or a transparent container for events, and the Subappication could be viewed as a transparent container for events and data or software tool construct. Thramboulidis, et al. [10] examined alternative means about scheduling. They especially focused on how to map the FBN to threads of operation system. On the other hand, formal methods were utilized or altered to accurately describe the execution process of the BFB or the FBN. Tu, et al. [11] proposed formal syntax and semantics of the BFB. Čengić, et al. proposed a formal framework for modeling Application [12] and execution semantics of the Application [13]. Many other formal methods such as net condition/event systems (NCES) and signal interpreted petri net (SIPN) were compared in [14].

The BFB is the essential construct of the standard. The CFB and the Application are simply represented as FBNs, whereas the BFB is declared by the interface, the ECC, the algorithms and the internal variables. Some semantic ambiguity relates to the interface and the ECC of the BFB, but few researches are focused on the improvement of definitions of those two elements. In this study, we propose a compact interface model and a strict ECC model to improve the type declaration of the BFB. The improved BFB contributes to reduce semantic ambiguity as well as to simplify the creation of a BFB and the usage of BFBs in a FBN.

The paper is organized as follows. The standard BFB type declaration is briefly introduced in Section 2. The compact interface model and the strict ECC model are presented in Section 3 and 4 respectively. A formal model of the improved BFB type declaration is addressed in Section 5. A BFB case study is presented in Section 6 and the paper is concluded in the Section 7.

2. Brief introduction to standard BFB type declaration

By the IEC 61499 standard, a BFB type is a named class that can be instantiated. The BFB type is declared by an interface, an ECC, zero or more algorithms, and zero or more internal variables, as shown in Fig. 1. The interface of a BFB consists of event inputs, event outputs, data inputs, and data outputs. Each event input has zero or more associated data inputs; each event output has zero or more associated data outputs. As shown Fig. 1 (a), the interface of the BFB named ADDITION consists of:

1) an event input named ADD of type EVENT;
2) an event output named DONE of type EVENT;
3) two data inputs named NUM1 and NUM2 of type INT; and
4) a data output named SUM of type INT.

An ECC consists of states and transitions. Each state has zero or more associated actions. Each of the actions associates the state with an algorithm or an event output, or both. Each transition has an associated transition condition which is an event input variable, a BOOL typed data variable, a guard condition, etc. As shown Fig. 1 (b), the ECC of the BFB named ADDITION consists of:

1) an initial state named START;
2) a normal state named STATE with an action associating the state with an algorithm ADDING and the event output DONE; and
3) two transitions: one is from START to STATE with the condition ADD, the other is from STATE to START with the condition TRUE.

The ECC controls the execution of a BFB. It plays an important role in combining the interface with the algorithms and the internal variables. The input events trigger the execution of the algorithms; the data input/output variables and internal variables are dealt by the algorithms; and the output events are issued on the completion of the algorithms.

The algorithms implement operations of a BFB. They can be programmed in various programming languages. As shown in Fig. 1 (c), the algorithm ADDING of the BFB named ADDITION is programmed in Structured Text. It adds two numbers and then produces a result.

Multiple instances of a BFB type can be generated. Each instance has a different name that is used for recognition and reference. The instances of the same BFB type have the same data structures and functions, but the instances are independent objects that have separate memory to store data and states. The instances are used in a FBN in a transparent way. Only the interface is visible whereas the ECC, the algorithms, and the internal variables are hidden.
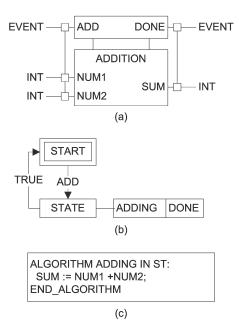

Fig. 1.   The example of a standard BFB type declaration. (a) Interface, (b) ECC, and (c) algorithm.

3. Compact interface model

When a BFB receives an input event, it is prerequisite that the associated data input variables are already prepared for being sampled. In other words, before an output event is issued, the associated data output variables must be got ready. The aforementioned facts imply that although event interface (including event inputs and event outputs) and data interface (including data inputs and data outputs) are implemented separately, they are tightly coupled. In this section, we propose a compact interface model to get rid of some semantic ambiguity related to such as when to sample data input variables, to decrease the number of connections of a FBN, and to facilitate the operation of BFBs in a FBN.

3.1. Structured event type declaration

It states in the standard that the event type is implicitly declared by its use and the event outputs can only be connected to the event inputs of compatible types. The event type adjacent to an event

input/output indicates which type of event that the event input/output can receive or emit. In the FBDK [15] implementation, the event type is arbitrarily defined as an identifier. We further develop the event type mechanism to implement a compact BFB interface. In our FBStudio implementation, the event type is declared as the structured type so that event is a composite object where the event and its associated data variables are encapsulated. Fig. 2 (a) shows an functional equivalent implementation of BFB ADDITION in the FBStudio. The event type ET_INT2 is declared to encapsulate an event with two data of type INT; the event type ET_INT is declared to encapsulate an event with one datum of type INT. Thereafter, the ET_INT2 and ET_INT are used to declare the event input ADD and the event output DONE respectively. In this way, the data interface is integrated into the event interface so that the explicit data interface is delimited.
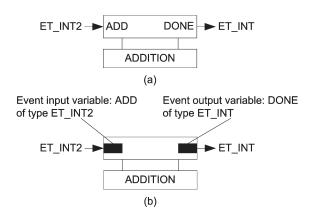

Fig. 2.   Illustration of event interface and event variables due to the structured event type declaration. (a) Interface and (b) event variables.

An event variable is implicitly declared for each event input/output. The type and name of an event variable are the same as that of the corresponding event input/output (Subclause 5.1 [3]), as shown in Fig. 2 (b) for the BFB ADDITION. When an event is received by a BFB through one of its event inputs, the event will be stored in the corresponding event input variable. Therefore, the associated data input variables are stored simultaneously into the event input variable as data members so that explicit sample is not needed. Before an output event is issued, it is prepared in the event output variable where data members are assigned by the corresponding algorithm. Therefore, when the output event is issued, its associated data output variables are sent out simultaneously as data members. To sum up, the compact interface model enables the event and its associated data variables to be received, issued, transferred, and buffered as a unique entity. Within the BFB the event variable is used instead of the event so that data members can be extracted to be used in the guard conditions and the algorithms.

As illustrated in Fig. 3 (a), the BOOL typed member variable State indicates the liveness of an event. When an event input variable is used as a transition condition, the TRUE of State will clear the transition and the FALSE of State will not. The STRING typed member variables Instance and Event are names of the BFB instance and the event output of the BFB. Those two variables identify an event and are used to facilitate event dispatch. Zero or more data members locate after the State member. Other members for special application purposes may be appended after the data members. When declaring an event type, its members can be initialized to appropriate values as shown in Fig. 3 (b) and Fig. 3 (c) in Structured Text.

Instance : STRING;
Event : STRING;
State : BOOL;
Data1;
Data2;
Datan;
Extended members;

(a)

```
TYPE ET_INT:
   STRUCT
      Instance: STRING := " ";
      Event: STRING := " ";
      State: BOOL := FALSE;
      Data: INT := 0;
   END_STRUCT;
END_TYPE
```

(b)

```
TYPE  ET_INT2:
   STRUCT
      Instance: STRING := " ";
      Event: STRING := " ";
      State: BOOL := FALSE;
      Data1: INT := 0;
      Data2: INT := 0;
   END_STRUCT;
END_TYPE
```

(c)

Fig. 3.   Event type declaration. (a) Structure of event type, (b) type declaration of ET_INT and (c) type declaration of ET_INT2.

3.2. Graphic notion

Most implementations support the movement of BFBs in a FBN. We additionally support the rotation of BFBs. With the help of the compact interface and the rotation facility, it is easy to get a clearer overview of the FBN, as illustrated in Fig. 4.

To facilitate event connections between rotated BFBs, we introduce arrow notation to event interface. As shown in the Fig. 2 (a), for the event input the arrow points into the BFB and for the event output the arrow points out of the BFB. Therefore, no matter what angle a BFB stands at, the event inputs and the event outputs can be easily recognized and discriminated. The event connections are then represented as directed links. That special improvement is inspired by the 4DIAC implementation [16] but the 4DIAC does not support rotation and arrows only exist in the FBN other than the BFB interface.
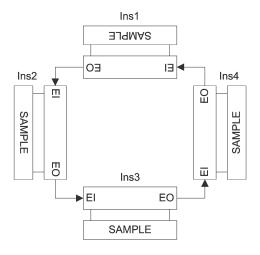
Fig. 4.   Illustration of rotation of BFBs.

3.3. Event conversion and buffer

Type conversion BFBs are needed to connect event outputs to event inputs of incompatible event types, e.g. connecting an event output to an event input with different data members. Fig. 5 shows an Application which adds four numbers and produces a result. The main idea is to use three instances of ADDITION. The Ins1 adds two input numbers; the Ins2 adds the other two input numbers; and the Ins3 adds the results of Ins1 and Ins2. Thus the result of Ins3 is the sum of the four input numbers. However the two event outputs of Ins1 and Ins2 cannot be directly connected to the only event input of Ins3. A conversion BFB is introduced to solve the problem. The BFB Conversion of type INTS_INT2 has two event inputs of type ET_INT and one event output of type ET_INT2. The earlier issued event DONE of Ins1 or Ins2 is buffered by the Conversion. The Conversion keeps waiting until another event DONE of Ins2 or Ins1 is issued. It then converts the two ET_INT typed input events to an ET_INT2 typed output event. Thereafter, an event EO will be issued by the Conversion to cause the Ins3 to be executed.
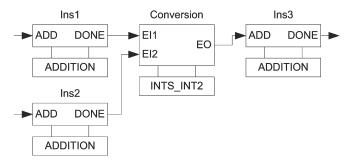

Fig. 5.   Applying the type conversion BFB in a FBN.

3. Strict ECC model

The standard leaves much freedom to implement an ECC as shown in Table I of FBDK implementation. In our FBStudio implementation some rules are added to implement a more practical and strict ECC and to decrease semantic ambiguity. The main idea is to simplify the transitions and conditions and move the guard condition from the transition to the action.

Table 1

FBStudio implementation of ECC compared with the FBDK implementation.

Item	#	FBStudio implementation	FBDK implementation
State	1	The ECC must have exactly one initial state and at least one normal state.	The ECC may have no states, only one initial state, or one initial state along with one or more normal states.
Transition	2	From a state to the same state, at most one transition can exist.	From a state to the same state, more transitions can exist
	3	Between any two different states, at most two transitions can exist and must be in the opposite direction.	Between any two different states, one or more transitions can exist.
	4	Every normal state must act as the source state and the destination state respectively at least once in transitions.	A state may or may not act as the source state or the destination state.
Transition	5	The transition condition must be either an	A transition condition may be an event

condition		event input variable (not the State member of an event input variable) or the Boolean constant TRUE.	input variable, the Boolean constant TRUE, a guard variable or a guard condition as well as an event input variable logically AND the guard variable or the guard condition.
	6	If the source state and the destination state of a transition are the same, the condition for the transition must be an event input variable.	If the source state and the destination state of a transition are the same, the condition for the transition must not be TRUE.
	7	If more than one transition starts with the same source state, conditions for those transitions must be event input variables and those event input variables must be different.	If more than one transition starts with the same source state, conditions for those transitions are not constrained
	8	The Boolean constant TRUE can be used as transition condition unless no cycle is caused.	The Boolean constant TRUE can be used as transition condition unless no cycle is caused.
	9	An action consists of three elements for the guard condition, the algorithm, and the event output respectively. The guard condition controls the execution of the algorithm and the emission of the output event that are referenced by the same action.	An action consists of two elements for the algorithm and the event output respectively.
Action	10	A guard condition can be one of 1) the Boolean constant TRUE; 2) a single BOOL typed internal variable or a single BOOL typed data member of event input/output variables; 3) a Boolean expression utilizing one or more internal variables and data members of event input/output variables.	A guard condition can be one of 1) the Boolean constant TRUE; 2) a single BOOL typed internal variable or input/output variable; 3) a Boolean expression utilizing one or more internal variables and input/output variables.

Those rules are classified into four groups corresponding to the four elements of the ECC. The rules are explained in detail as follows.

4.1. State

As an ECC with no states or only one initial state cannot do any meaningful work, we apply the rule #1. It is noted that the standard definitely states that the initial state shall have no associated actions

4.2. Transition

Rule #2 and #3 are applied to eliminate redundant transitions whereas rule #4 is applied to avoid deadlock states where no transitions can clear. Fig. 6(a) shows an ECC conflicting with those rules, whereas Fig. 6(b) shows an ECC following those rules. The #number in Fig. 6(a) indicates with which rule the adjacent transition conflicts.
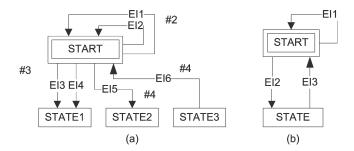
Fig. 6. Illustration of ECC transition rules. (a) ECC with incorrect transitions and (b) ECC with correct transitions.

4.3. Transition condition

Compared with the standard or the FBDK implementation, our implementation is highly simplified as indicated in rule #5. In general, event input variables are preferred and in some cases are mandatory as indicated in rule #6 and #7. The Boolean constant TRUE may be used as condition but cycle should be avoided as indicated in rule # 8. It is noted that the rule #7 solves the priority problem [7] as long as the input events are passed to the BFB sequentially. The deal lock problem related to the conditional arc [8] is also got rid of because guard conditions are not used as transition conditions. Fig. 7(a) shows an ECC conflicting with the rule #8, whereas Fig. 7(b) shows two ECCs following the rule #8. In Fig. 7(a), the two Boolean constants TRUE form an infinite cycle between the two states START and STATE. That is illegal situation in our FBStudio implementation as that in the FBDK implementation.
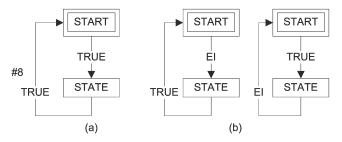

Fig. 7. Illustration of ECC transition condition rules. (a) ECC with incorrect transition conditions and (b) ECCs with correct transition conditions.

4.4. Action

In our FBStudio implementation, the action is extended by adding another element, i.e. the guard condition as indicated in rule #9. The guard condition controls whether the action is to be executed or not. The guard condition in our FBStudio implementation only uses internal variables and data members of event variables as indicated in rule #10. Fig. 8 shows an ECC with actions. The algorithm and the event output are optional, but an action should at least have one of them to perform some meaningful work. The guard condition must exist in any actions. It should be evaluated to BOOL typed values TRUE or FALSE. If the guard condition is TRUE, the action will be executed; if the guard condition is FALSE, the action will not be executed.
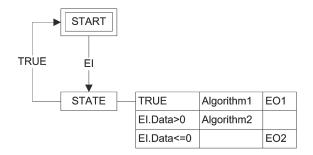
Fig. 8. Illustration of ECC action rules.

5. Formal modeling

Formal models for the standard BFB were presented in [11][12][17]. In this section, we develop a formal model for the improved BFB. We only focus on the formal modeling of the BFB type declaration other than the execution semantics.

5.1. Formal description

A BFB type is a 5-tuple defined as

$$BFB = (EI, EO, ECC, ALG, VAR)$$

where

$EI = \{ei_1, ei_2, ...\}$ is a set of event inputs;

$EO = \{eo_1, eo_2, ...\}$ is a set of event outputs;

$ALG = \{alg_1, alg_2, ...\}$ is a set of algorithms;

$VAR = \{var_1, var_2, ...\}$ is a set of internal variables.

The ECC in turn is a 6-tuple defined as

$$ECC = (S, T, A, C, f_A, f_C)$$

where

$S = \{s_0, s_1, s_2, ..., s_{m-1}, m \geq 2\}$ is a set of EC states and s_0 is the initial state;

$T \subseteq S \times S$ is a set of arcs representing EC transitions;

$A \subseteq G \times (ALG \cup EO \cup ALG \times EO)$ is a set of EC actions and G is the guard condition;

$G \subseteq \{TRUE\} \cup$

    $\{ei \in EI \mid BOOL$ typed data members of $ei\} \cup$

    $\{eo \in EO \mid BOOL$ typed data members of $eo\} \cup$

    $\{BOOL$ typed member of $VAR\} \cup$

    $\{Boolean$ expression on member of VAR, ei, and eo$\}$

$C \subseteq \{TRUE\} \cup EI$ is a set of transition conditions;

$f_A : S \rightarrow A$ is the function assigning sequences of EC actions to the states;

$f_C : T \rightarrow C$ is the function assigning conditions to transitions.

5.2. Formal verification

Based on the formal model, the ECC rules of Table 1 are checked as follows.

1) Rule #1 is guaranteed by the set definition of S; rules #2 and #3 are guaranteed by the set definition of T.

2) To check rule #4, the following terms are defined.

$Source(s) = \{s \in S \mid (s,s_i) \in T, i=0,...,m-1\}$ is the set of transitions that start with the state s.

$Destination(s) = \{s \in S \mid (s_i,s) \in T, i=0,...,m-1\}$ is the set of transitions that end with the state s.

Then the rule #4 requires that both $|Source(s)| > 0$ and $|Destination(s)| > 0$ are TRUE for every normal state $s \in S$.

3) The rule # 5 is guaranteed by the set definition of C.

4) $t \in Source(s)$ and at the same time $t \in Destination(s)$ indicates $t = (s, s)$, i.e. the source state and the destination state of the transition t are the same state s. According to the rule #6, the condition for the transition t must be an event input variable.

5) When $|Source(s)| > 1$, the rule #7 requires that conditions for transitions $t \in Source(s)$ must be different event input variables.

6) For each subset $P \subseteq S$, if transitions exist to sequentially link all the states of P, we define a path that exists in the P. The rule #8 requires that the conditions of the transitions of every closed path are not all TRUE.

7) The rule #9 and the rule #10 are guaranteed by set definition of A and G respectively.

6. Case study

A BFB is designed to perform both integer division and floating-point division. When the BFB is implemented in the FBDK, the interface consists of two event inputs, two event outputs, four data inputs, and two data outputs as shown in Fig. 9(a). In addition, the ECC can be implemented in a number of ways and the Fig. 9(b) shows one possibility. The lower branch for integer division is safe no matter whether the divisor INUM2 is zero. The upper branch for floating-point division is safe only when the divisor FNUM2 is not zero, but when FNUM2 is zero the ECC will be frozen in the state of FDIV, which indicates using guard condition alone as transition condition may cause potential problem [7][8].
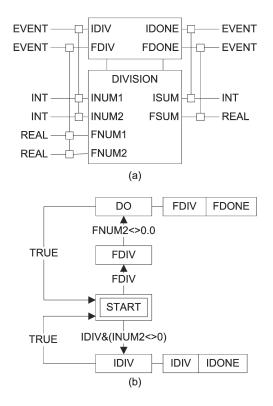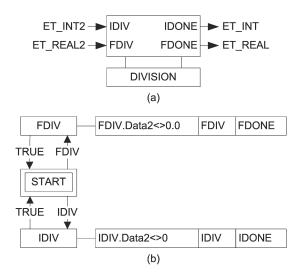
Fig. 9.   FBDK implementation of the BFB DIVISION. (a) Interface and (b) ECC.

The equivalent function can be implemented in the FBStudio as shown in Fig. 10. ET_REAL2 has two REAL typed data members Data1 and Data2, and the ET_REAL has only one REAL typed data member Data. Compared with the FBDK implementation, advantages of FBStudio implementation are:

  1) the interface only consists of two event inputs and two event outputs, i.e. the content of interface decreases up to 60%;

  2) the strict rules for transitions and transition conditions lead to a rather intuitive decision process when design an ECC;

  3) the conditional arc is illegal in the strict ECC model so deadlock states can be avoided ; and

  4) shifting guard conditions to actions enables the strict ECC as powerful as that of the FBDK implementation.

Fig. 10. FBStudio implementation of the BFB DIVISION. (a) Interface and (b) ECC.

7. Conclusion

We improve two elements of the IEC 61499 BFB, i.e. the interface and the ECC. We propose a compact interface model to integrate the data interface into the event interface so that data interface can be eliminated; the related semantic ambiguity can be got rid of; and the number of connections can be decreased. We also propose a strict ECC model to add some practical rules to the standard. Those rules relate to the ECC elements: state, transition, transition condition, and action. That strict ECC model reduces design redundancy and gets rid of some semantic ambiguity. In addition, we introduce arrow notation to event interface, which helps to maintain clear connections of a FBN even if some FBs are rotated to an arbitrary angle. In our future work, type declaration of CFB and SIFB will be improved to be consistent with the improved BFB. We have been developing a tool called FBStudio to construct FB Applications, in which the improved type declaration of the BFB is implemented. An embedded executable environment called FBController is also under development. With the FBStudio and the FBController, embedded controllers are hoped to be fully customized and rapidly implemented in order to respond challenging requirements of industry.

References

[1] J. Wan, M. Chen, F. Xia, D, Li, and K. Zhou, "From machine-to-machine communications towards cyber-physical systems," Computer Science and Information Systems, vol. 10, no. 3, pp. 1105-1128, 2013.

[2] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in IEEE International Conference on Wireless Communications and Signal Processing (WCSP), Nanjing, CN, 2011, pp. 1-6.

[3] Function blocks-Part 1: Architecture, IEC 61499-1, 2005.

[4] M. V. Garcia, F. Perez, I. Calvo, and G. Moran, "Building industrial CPS with the IEC 61499 standard on low-cost hardware platforms," In IEEE Emerging Technology and Factory Automation (ETFA), Barcelona, ES, 2014, pp. 1-4.

[5] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," IEEE Trans. Ind. Informat., vol. 7, no. 4, pp. 768-781, Nov. 2011.

[6] V. Vyatkin, "The IEC 61499 standard and its semantics," IEEE Ind. Electron. Mag., vol. 3, no. 4, pp. 40-48, Dec. 2009.

[7] V. Vyatkin and V. Dubinin, "Sequential axiomatic model for execution of basic function blocks in IEC61499," in Proc. 5th IEEE Int. Conf. Ind. Informat., Vienna, AT, 2007, pp. 1183-1188.

[8] V. Vyatkin and V. Dubinin, "Refactoring of execution control charts in basic function blocks of the IEC 61499 standard," IEEE Trans. Ind. Informat., vol. 6, no. 2, pp. 155-165, May 2010.

[9] C. Sünder, A. Zoitl, J. H. Christensen, M. Colla, and T. Strasser, "Execution models for the IEC 61499 elements composite function block and subapplication," in Proc. 5th IEEE Int. Conf. Ind. Informat., Vienna, AT, 2007, pp. 1169–1175.

[10] K. Thramboulidis, G. Doukas. "IEC61499 Execution Model Semantics," in Innovative Algorithms and Techniques in Automation, Industrial Electronics and Telecommunications, Heidelberg: Springer Netherlands, 2007, pp. 223-228.

[11] Y. Tu, D. Li, and S. Li, "Formal syntax and semantics of basic function blocks in IEC 61499," Proc. Institution Mech. Engineers, Part C: J. Mech. Eng. Sci., vol. 226, no. 4, pp. 1025-1035, Apr. 2012.

[12] G. Čengić and K. Åkesson, "On Formal Analysis of IEC 61499 Applications, Part A: Modeling," IEEE Trans. Ind. Informat., vol. 6, no. 2, pp. 136-144, May 2010.

[13] G. Čengić and K. Åkesson, "On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics," IEEE Trans. Ind. Informat., vol. 6, no. 2, pp. 145-154, May 2010.

[14] G. Frey and T. Hussain, "Modeling techniques for distributed control systems based on the IEC 61499 standard - Current approaches and open problems," in Proc. 8th Int. Workshop Discrete Event Syst., Ann Arbor, US, 2006, pp. 176-181.

[15] FBDK software, [Online]. Available: www.holobloc.com, 2015.

[16] 4DIAC software, [Online]. Available: www.fordiac.org, 2015.

[17] V. Dubinin and V. Vyatkin, "Towards a formal semantic model of IEC 61499 function blocks," in Proc. 4th IEEE Int. Conf. Ind. Informat., Singapore, SG, 2006, pp. 6-11.

Shiyong Wang was born in Huoqiu County, Anhui Province, China, in 1981. He received the B.S. and Ph.D. degrees in mechanical & electrical engineering from South China University of Technology, Guangzhou City, Guangdong Province, China, in 2010.

Since 2010, he has been a Lecturer with the Mechanical & Electrical Engineering Department, South China University of Technology. He is the author of more than 10 articles and holds four patents. His research interests include motion control, robotics, and high-performance embedded control system.

Mr. Wang was a recipient of the First Prize for Science & Technology Development of Guangdong Province in 2009.

Chunhua Zhang was born in Pulandian City, Liaoning Province, China, in 1976. She received the B.S. degree in material processing engineering from Nanchang Hangkong University, Nanchang, China, in 1999, and the M.S. degree in material processing engineering and Ph.D. degree in mechatronic engineering from South China University of Technology, Guangzhou City, Guangdong Province, China, in 2002 and 2014 respectively. Her research interests include embedded and networked control system, and motion control system.

Dongyao Jia received the B.E. degree in automation from Harbin Engineering University, Harbin, China, in 1998, the M.E. degree in automation from Guangdong University of Technology, Guangzhou, China, in 2003, and Ph.D. degree in computer science from City University of Hong Kong in 2014. He is currently a Research Fellow in Institute for Transport Studies (ITS), University of Leeds, UK. He was a visiting scholar in University of Waterloo in 2014. He worked as a senior engineer in the telecom field in China from 2003 to 2011. He also took part in the establishment of several national standards for home networks. His current research interests include vehicular cyber-physical systems, traffic flow modeling, and internet of things.