Automatic Model Generation of IEC 61499 Function Block Using Net Condition/Event Systems

Cheng Pang, *Non-Member*, Valeriy Vyatkin, *Senior Member IEEE*
The University of Auckland, New Zealand
cpan024@ec.auckland.ac.nz, v.vyatkin@auckland.ac.nz

***Abstract*-The IEC 61499 standard establishes a framework specifically designed for the implementation of decentralized re-configurable industrial automation systems. However, the process of distributed system's validation and verification is difficult and error-prone. This paper discusses the needs of model generators which are capable of automatically translating IEC 61499 function blocks into formal models following specific execution semantics. In particular, this paper introduces the prototype Net Condition/Event Systems model generator and aims to summarize the generic techniques of model translation.**

I. Introduction

With the increasing use of decentralization and the growing complexity in industrial automation systems, a formal and systematic verification of the system is essential for revealing subtle design pitfalls and for avoiding system failures. Traditionally, hardware systems are validated by the process of simulation and testing. A system is first simulated and then tested on a prototype. This approach is effective in the early debugging stage, but it quickly becomes inefficient and inadequate when the design complexity increases as the growing potential of defects necessitates the use of ever more testing scenarios. More importantly, the traditional approach is a very time-consuming process requires sophisticated software programs and it only guarantees the correct behaviours in the tested scenarios. As *Programmable Logic Controllers* (*PLC*'s) and *IEC 61131-3* [1] are still the mainstream re-programmable electronic devices and the corresponding programming standard in industrial automation systems, the conventional system verification process is relatively intuitive. However, *PLC* programs' centralized control model, cyclic execution semantics, and lack of interoperability significantly restrict the software's reusability and the system's re-configurability.

To overcome the shortcomings of *PLC*, the *International Electrotechnical Commission*, *IEC*, developed the new international standard *IEC 61499* [2], which inherits the function block concept from *IEC 61131-3* and establishes an open, modularized, and implementation-independent software development framework to satisfy the current industrial needs for agile development of interoperable, re-configurable, and portable distributed automation systems. However, as *IEC 61499*-compliant applications can be highly decentralized and of a large scale, the traditional validation approaches are insufficient and inefficient to provide an adequate analysis of system properties.

As an alternative to the simulation and testing approach, *formal verification* [3] exhaustively explores all possible behaviours of the target systems. By choosing a proper formal method, it is able to automatically verify all the interested system properties and hence identify potential design pitfalls. Therefore, as discussed in several papers [4-7], verifying function block's formal model is an effective way to validate the original design. However, the semantic ambiguities of *IEC 61499* lead to different interpretations of the standard and directly result in a variety of standard-compliant function block implementations with contrary execution behaviours [8-11]. It is then realized that formal modelling of function blocks needs to be preceded by the definition of their complete execution semantics [5]. Discussion of various execution models is not within the scope of this paper; rather, this paper aims to present a prototype model generator which translates function blocks into *Net Condition/Event Systems* (*NCES*) [11] models following the *sequential execution model* [9].

The paper is structured as follows. Section II briefly reviews existing approaches of modelling *IEC 61499* function block and gives a short comparison. Section III outlines the overall methodology of modelling function blocks using *NCES* and the details of modelling *IEC 61499* entities are elaborated in Section IV. Then, Section V generally discusses the verification framework and the properties need to be verified. Finally, the paper concludes with outlook of future works.

II. Modelling of IEC 61499

In the past few years, numerous modelling approaches for the *IEC 61499* have been introduced. Besides the different formalisms and verification techniques employed in these approaches, they also differ in the level of model abstraction and the emphasis in the model analysis. This section outlines some different ways of modelling *IEC 61499* and then gives a brief comparison between the *NCES* modelling approach and the others.

A. Interacting Finite Automata Modelling Approach

Čengić et al. developed a Java-based runtime environment, called *FUnction Block Execution Runtime* (*Fuber*) [12], for the execution of *IEC 61499* applications. *Fuber* implements a sequential scheduling function block execution model using a *First-In-First-Out* (*FIFO*) queue. Each *IEC 61499* application in *Fuber* is translated into a set of interacting finite automata by modelling the *FIFO* queue, the event execution semantics, basic function blocks, service function blocks, composition function blocks, and the connections between function block instances within the application. The resultant model is verified using the supervisory control theory [13].

B. *Timed Automata Modelling Approach*

In [7], Stanica et al. proposed a modular translation approach for modelling *IEC 61499* function block behaviour in timed automata. In their work, the primary concern is the execution control of function blocks, especially their external behaviours. As a result, algorithms and data of basic function blocks are not considered. Instead, algorithms are modelled by their processing time and data are abstracted. Each basic function block is modelled by a set of automata: an *Execution Control Chart* (*ECC*) automaton, the corresponding event input automata, and two additional synchronization automata. Modelling of a function block network is accomplished by direct composition of function block models scheduled by a scheduling automaton. The scheduling automaton implements a non-pre-emptive and non-prioritized scheduling policy. The timed automata are verified by using the *UPPAAL* [14] tool.

C. *UML Modelling Approach*

Differing form other modelling approaches, Panjaitan et al. [15] presented a way of using the *Unified Modelling Language* (*UML*) to model not only *IEC 61499* applications, but also the entire development process of distributed control systems. In this approach, every development step is depicted in *UML* and then the actual system design is transformed into an *IEC 61499* model according to the *UML* representation. The *UML* model, however, does not follow any particular execution semantics. The operation sequence of function block instances inside an application, for example, is specified using a *UML* sequence diagram. Therefore, the transformed function block's execution model is completely dependent on the sequence diagram.

D. *Prolog Modelling Approach*

Dubinin et al. introduced a new way of modelling *IEC 61499* function block networks by using the logic programming language *Prolog* [16] in [6], where closed-loop function block systems with arbitrary data types are modelled and verified. The *Prolog* model uses production rules [16] to represent function block networks, which allow concurrent execution of constituent function block instances. Each *IEC 61499* entity is directly mapped to a *Prolog* term with the same tag name and algorithms are represented in predicates. The liveness and safety properties of the modelled systems are then specified *Prolog* predicates and verified.

E. *NCES Modelling Approach*

NCES is a distributed state formalism developed for modelling discrete-event systems. It inherits the graphical notation and non-interleaving semantics of classic *Petri-net* and supports the notion of modularization. The event-driven, modular design and hierarchical structure of *NCES* closely correspond to the design methodology of *IEC 61499* and make the modelling process intuitive.

NCES was first used to model the execution logic of a basic function block in [4] by Vyatkin and Hanisch, and later [8] studied the *NCES* model of a function block network following a parallel execution semantics. In general, each *IEC 61499* is modelled by a separate *NCES* module. The individual modules are then assembled or composed, in *NCES*'s terminology, following the same topological structure of *IEC 61499*. The operation of a basic function block, on the other hand, is controlled by an *NCES* module implementing the *Execution Control Operation State Machine* (*ECOSM*) defined in the standard.

Compared to other methods, the *NCES* modelling approach has the following advantages:

- Unlike automata based modelling approaches, *NCES* modules directly model the actual operations of function block systems. The hierarchical structure similar to that in the standard can help identify not only in which states the system fails, but also which function block fails. Moreover, the existing *NCES* modules can be reused which makes progressive modelling possible.
- Contrary to the *Prolog* approach, the model's execution semantics does not depend on the modelling formalism. Although *NCES* per se is a concurrent formalism, by sequentially queuing the event signals it is also able to model sequential behaviours.
- *NCES* models can be verified by using the *model checking* [3] technique, which supports unsupervised automatic verification and identifies system failure via counterexamples.

III. Modelling Methodology Outline

In *IEC 61499*, function blocks are the basic building units. Composition of function block instances forms higher level entities such as *resources* inside *devices*, which ultimately constitute a *system*. This hierarchical design topology implies a corresponding bottom-up modelling approach: simplest functional components are modelled first and eventually model the entire system by assembling and integrating the basic models. The bottom-up modelling approach can ensure that higher level components are always built on top of lower level elements whose models have already been validated. Moreover, optimization of the resultant model can be achieved by optimizing the constituent models. In this *NCES* modelling framework, basic function blocks are modelled first and then gradually extend the framework to model the entire system.

The essential of modelling is to abstract the properties of the subject system while preserving its nature and reflecting its characteristics. In order to closely model the target function block system and to establish a basis for decision-making during the modelling process, the following principles are set:

- Strictly comply with the sequential execution model: the *NCES* models must closely follow the postulates defined in the sequential execution model. If exact modelling is impossible or impractical, the model must therefore be behaviourally equivalent to the system being modelled.
- Model as many details as possible: modelling the operation details of *IEC 61499* function blocks enables verification tools to identify more concrete design pitfalls.

- Optimize and verify the model after creation: in order to reduce the state space and to ensure the correctness of the final model, each sub-model is optimized and verified.

On the other hand, as the sequential execution model requires a *FIFO* queue to coordinate events in a function block network, an extra *NCES* module implements an *Event Dispatcher* must be designed to sequentially schedule, for example, event forking and merging among function blocks.

IV. Modelling IEC 61499 Function Block

A. *Modelling of Basic Function Block*

Basic function block is the fundamental construction unit in *IEC 61499* architecture, which encapsulates the functionality description in its *ECC*, and communicates with the external environment through its input and output ports defined in its *interface*. The internal operations of a basic function block are coordinated by its *ECOSM*. Fig. 1 shows a simple basic function block *UpdateX* whose sole purpose is to update the Boolean output *QO* according to the value of Boolean input *X*.

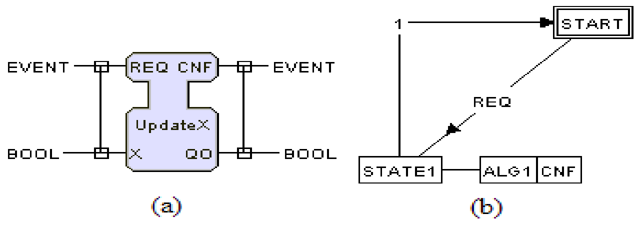

Fig. 1. Simple Basic Function Block: (a) Interface, (b) ECC

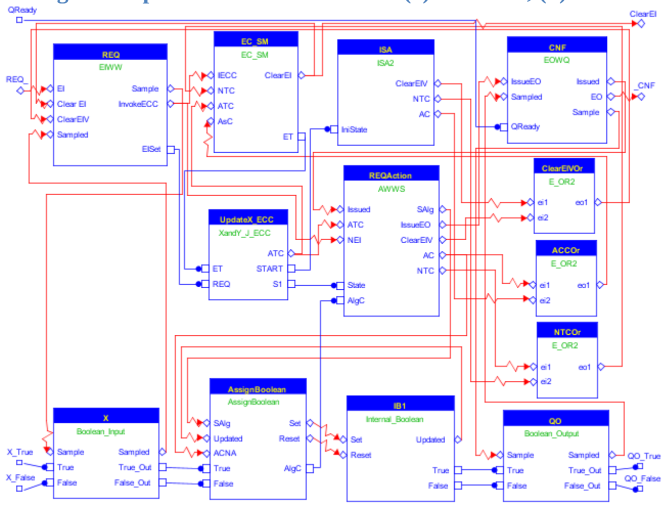

Fig. 2. Translated NCES Model

Following the modelling methodology outlined in Section III, each functional component is modelled by an *NCES* module. As illustrated in Fig. 2, *REQ* and *CNF* modules model the event ports, *X* and *QO* modules model the data ports, and *EC_SM* module models the *ECOSM*. Moreover, the *WITH* association between event and data port pairs are accomplished via the *Sample* and *Sampled* signals between the corresponding *NCES* modules. For example, when the event signal *REQ_* signal arrives, the *REQ.Sample* signal will be issued to trigger the sampling process of *X* module. When *X* updated the Boolean value it represents, signal *X.Sampled* will be sent back to *REQ* module. Modelling of event/data ports, *ECOSM*, and simple algorithms has been studied in [5] and [4], which are still applicable in this sequential modelling approach. Therefore, the rest of this section will focus on the modelling of *ECC* and briefly describe the translated model's operations.

In general, the complete model of an *ECC* consists of three parts: the model of the *ECC* control flow, the model of *EC transitions*' Boolean condition, and the model of *EC actions*, interconnected via event and condition signals. As indicated in Fig. 2, the *UpdateX_ECC* module models the control flow of the *ECC* shown in Fig. 1 (b) and *REQAction* module models the *EC action* associated with the *EC state STATE1*. Since the *EC transitions* in this example only involve simple Boolean conditions, the Boolean conditions are modelled directly within the *UpdateX_ECC* module as illustrated below:

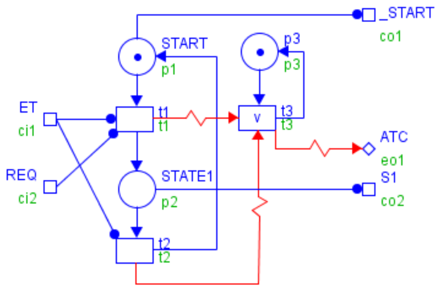

Fig. 3. UpdateX_ECC Module

Structurally, the model of the *ECC* control flow mimics the original *ECC*. Each *EC state* is mapped to an *NCES place* and each *EC transition* is transformed into an *NCES transition*. Any trivial Boolean condition is modelled by a set of condition signals connected to the target transition. In addition, the initial state is modelled by the initial marking of the *NCES* model. As shown in Fig. 3, the *EC state START* and *STATE1* are respectively mapped to the *NCES place p1* and *p2*. Similarly, the *EC transition START→STATE1* and *STATE1→START* are modelled by the *NCES transition t1* and *t2* respectively. At last, the token in *p1* marks it as the initial state.

According to the types of succeeding *EC transitions*, the number of *EC actions* an *EC state* has and whether the *EC action* associated with algorithm and event output, there can be 24 different types of *EC actions*. Therefore, 24 *NCES* modules are created to model all these different *EC actions*. Apart from the different module content, the main purpose of these NCES models is to supervise the execution of the associated algorithm module and the issuance of the event output. The *REQAction* module illustrated in Fig. 1 models a single *EC action* associated with an algorithm and an event output and its succeeding *EC transition* only contains a guard condition. Although the standard defines that no *EC action* should associate with the initial state, a helper module *ISA* is connected to the initial state in the *NCES* model. The function of *ISA* module is to clear the current registered event input in the initial *EC state* to avoid deadlocking, which does not affect the *ECC's* control flow.

The internal operations of the translated *NCES* model are as follows. Upon the arrival of the *REQ_* signal, *REQ* module triggers the sampling process of *X* module. Then, *EC_SM* module starts evaluating the *ECC* control flow inside the *UpdateX_ECC* module. When the *REQAction* module finishes executing the associated algorithm module *AssignBoolean*, it triggers the *CNF* module to update the data output. If the external event dispatcher module is ready to receive next event, the *QReady* signal will present and trigger the issuance of *_CNF* signal to complete the execution of the *NCES* model.

B. *Modelling of Event Dispatcher*

The sequential execution model requires that at any instant in time only one function block instance in a resource is active. Event signals in a composition function block are scheduled by an event dispatcher, whose primary task is to sequentially register the occurrences of event signals inside the function block network and then emit then orderly. The dispatcher must also ensure that no event signal will be lost and event will be emitted only if the previous event has been cleared. Thus, a *FIFO* queue is employed inside the dispatcher to store the event signals. Fig. 4 shows an event dispatcher model with three event inputs and a queue size of two:

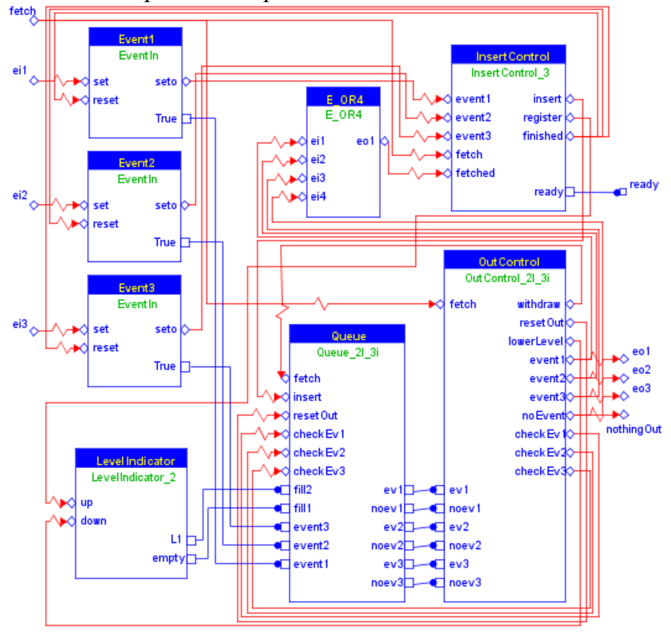

Fig. 4. Event Dispatcher

The event dispatcher can perform two interleaved operations: *event registration* and *event fetching*. When the *ready* event output is enabled, the event dispatcher is ready to register events occurring at *ei1*, *ei2*, or *ei3* once a time. The event signal will be stored in the *Queue* module and the inserting point is adjusted by the *LevelIndicator* module. On the other hand, the event fetching process is triggered by the external *fetch* signal. Firstly, the *InsertControl* module disables its *ready* output signal to signify the unavailability of the event dispatcher. Meanwhile, the *OutControl* module starts evaluating its condition inputs sequentially and decides which port to emit the first event stored in the *Queue* module. Then, the *LevelIndicator* module will restore the previous event inserting point. After the three modules, *Event1*, *Event2*, and *Event3* are reset and the *ready* output is re-enabled, the *OutControl* module emits the corresponding event signal to complete the fetching process. If no event is stored in the queue, the *nothingOut* signal from the *OutControl* module will be issued instead. The following figure illustrates the details of the *Queue* module:

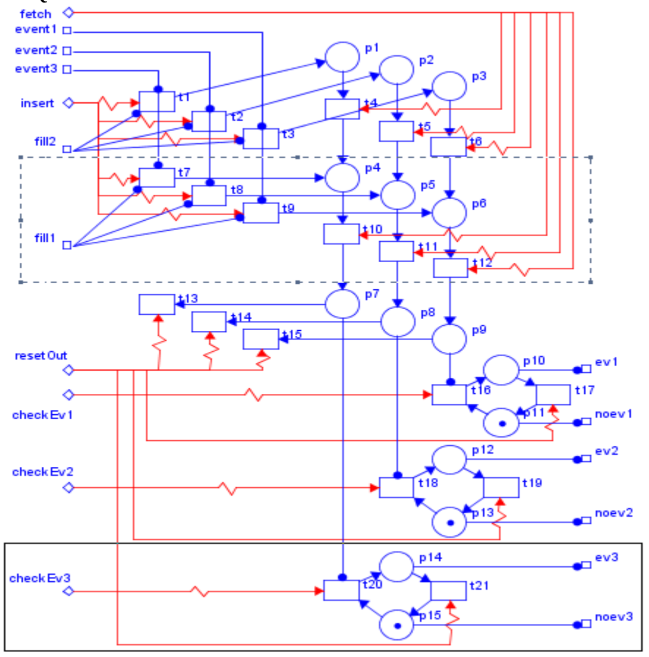

Fig. 5. FIFO Queue Model

The *Queue* module used in this example can queue two event signals from the three inputs: *event1*, *event2*, and *event3*. Initially the queue is empty. When the *insert* signal arrives, one of the three event signals will be stored as a token at the level indicated by *fill1* or *fill2* signals. During the fetching process, tokens inside *place p7*, *p8*, and *p9* will be cleared via *transition t13*, *t14*, and *t15* respectively and the corresponding *ev* output signal will be enabled. Moreover, the number of level and input can be increased by duplicating the respective structure inside the dot-line and solid-line squares.

C. *Modelling of Composite Function Block*

Unlike basic function blocks, before assembling the component models, the network inside the composite function block must be first analyzed to determine the size of the event dispatcher module and whether extra modules are inserted to fork or merge the event signals.

As the number of events a function block will emit depends on in which state the function block is and the sequence of state transition is unpredictable before the actual execution of the function block, currently it is assumed that the dispatcher's size is twice the total number of all function block instances' event outputs inside the network. The event dispatcher bridges the function block instances in the following way: all the instances' outputs are connected to the inputs of the dispatcher whose outputs are then fed to the respective instances' inputs.

Moreover, extra helper module, such as the standard defined *E_SPLIT* and *E_MERGE* function block modules, must be inserted accordingly to explicitly fork or merge the event signals.

Due to the hierarchical modelling approach, the translated *NCES* model is almost structurally identical to the original composite function block. As demonstrates in Fig. 6, apart from the event dispatcher module and the helper module *E_OR3*, the *NCES* model's layout and connections are identical to the original composite function block.

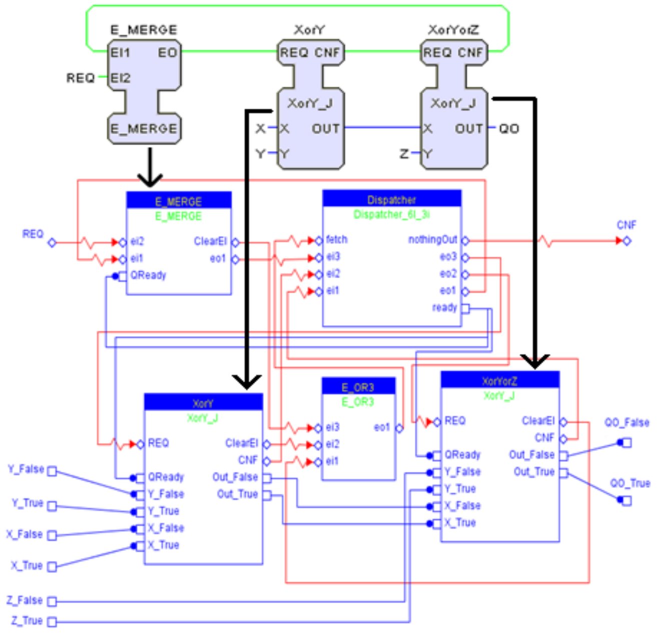

Fig. 6. Translated Composite Function Block Module

V. Property Verification

This section presents a verification framework borrowed from [19] for verifying *NCES* models and summaries general properties that must be valid in order to ensure the model is semantically and behaviourally correct. The function block to be verified is created in *FBDK* [17] and the translated *NCES* model can be opened in *ViEd* [18] and verified in *ViVe* [18].

In according to the sequential execution model, any *IEC 61499* application should possess two broad properties:

- All events will be processed, and
- Only one function block instance is active at any instant in time.

These two properties can be further refined according to the actual type of function block being investigated.

A. *Verification of Basic Function Block Properties*

Following the sequential execution model, a basic function block should have the properties listed below:

- *Guaranteed response*: the correct event output will eventually be issued after the occurrence of the corresponding event input.
- *All event inputs will be cleared*: any event input will be cleared regardless of whether or not it has been used in the *ECC* evaluation.

In order to verify a given function block model, an extra *event generator* module is created to generate all possible combinations of the input signals for the model being verified. The following basic function block is used to demonstrate the verification framework and to verify the properties listed above.

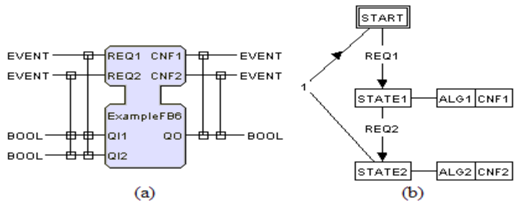

Fig. 7. ExampleFB6 Basic Function Block: (a) Interface, and (b) ECC

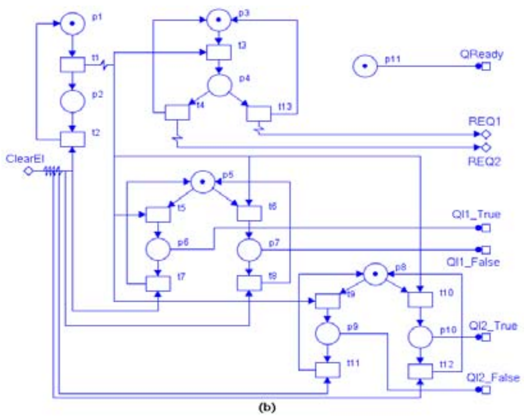

Fig. 8. EventGenerator1 Module

As indicated in Fig. 8, in order to generate all possible combinations of the four input signals of the *ExampleFB6* function block shown in Fig. 7, the *EventGenerator1* module heavily utilize conflicts in *NCES*. For example, in the above diagram, after the firing of transition t1, the token inside places *p5* can flow to either *p6* or *p7*. As a result, either *QI1_True* or *QI1_False* will be enabled. Moreover, since the verification of a single basic function block does not require an event dispatcher, place *p11* is used to simulate an empty queue.

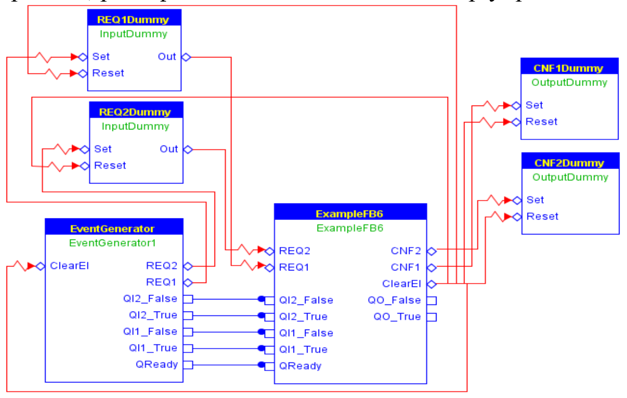

Fig. 9. Framework for Verifying ExampleFB6 Function Block

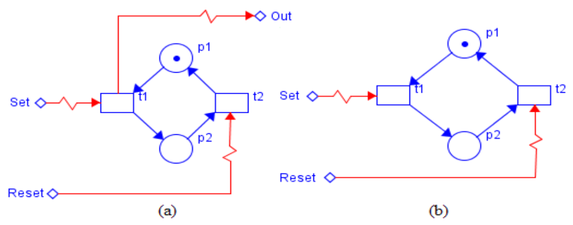

Fig. 10. Event Dummy Modules: (a) InputDummy Module, and (b) OutputDummy Module

Referring to Fig. 9, the verification framework consists of the event generator module, the target function block model, and the *event dummy* helper modules. The event dummy modules are simply bi-stables as shown in Fig. 10. Their sole purpose is to provide an indication of the event occurrence. For instance, in Fig. 9, the emission of the *EventGenerator.REQ1* event signal will move the token in *REQ1Dummy.p1* to *REQDummy.p2,* based on the token location, the occurrence of the event input can be easily identified. Therefore, to verify the event related properties, it is only necessary to check the markings of the event dummy modules.

According to the *ExampleFB6* function block's *ECC*, the *CNF1* event output is issued after the *REQ1* event input, whereas the *REQ2* event input is followed by the *CNF2* event output. Therefore, the following temporal logic statement is used to verify this guaranteed response property:

$$AG((P^{REQ1}\rightarrow EFP^{CNF1})\vee(P^{REQ2}\rightarrow EFP^{CNF2}))$$

where, predicate P^{REQ1} denotes the place *REQ1Dummy.p2*, predicate P^{REQ2} denotes the place *REQ2Dummy.p2*, predicate P^{CNF1} denotes the place *CNF1Dummy.p2*, and predicate P^{CNF2} denotes the place *CNF2Dummy.p2*. This statement means that in every state the occurrence of the event input *REQ1* or *REQ2* will eventually lead to the emission of the corresponding event output *CNF1* or *CNF2*. In ViVe, this property is proved to be true.

On the other hand, to verify that all event input will be cleared, the statement below is checked:

$$AG((P^{REQ1}\rightarrow AFP^{REQ1'})\vee(P^{REQ2}\rightarrow AFP^{REQ2'}))$$

where, $P^{REQ1'}$ denotes the place *REQ1Dummy.p1*and $P^{REQ2'}$denotes *REQ2Dummy.p1*. An event input is cleared when the tokens in the input dummy modules flow back to their initial places. The above statement is also proved to be true.

B. *Verification of Composite Function Block Properties*

In addition to the properties of basic function blocks, a composite function block must also ensure that no event signal will be lost and at any time instant there will be only one active function block instance. Since in our framework the event scheduling mechanism is explicitly implemented in the even dispatcher model, therefore the aforementioned properties are automatically satisfied.

VI. Conclusions

This paper presents a prototype model generator which intends to automatically translate *IEC 61499* function blocks into functionally and semantically equivalent *NCES* models following the sequential execution model. The translated formal models can then be systematically verified and analyzed by model checking tools.

To further extend current modelling architecture, future works would be applying the existing techniques to model function blocks obeying other execution semantics, such as scan-based or parallel execution model.

References

[1] International Electrotechnical Commission, *Programmable Controller - Part 3: Programming Languages, IEC 61131-3 Standard*. Geneva: International Electrotechnical Commission, 1993.

[2] International Electrotechnical Commission, *Function blocks for industrial-process measurement and control systems - Part 1: Architecture*. Geneva: International Electrotechnical Commission, 2005.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge: The MIT Press, 1999.

[4] V. Vyatkin and H. M. Hanisch, "A modeling approach for verification of IEC1499 function blocks using net condition/event systems," in *7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '99)*, Barcelona, Spain, 1999, pp. 261-270 vol.1.

[5] C. Pang and V. Vyatkin, "Towards Formal Verification of IEC 61499: Modelling of Data and Algorithms in NCES," in *5th IEEE Conference on Industrial Informatics (INDIN 2007)*, Vienna, Austria, 2007.

[6] V. Dubinin, V. Vyatkin, and H.-M. Hanisch, "Modelling and Verification of IEC 61499 Applications using Prolog," in *11th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2006)*, Prague, Czechoslovakia 2006, pp. 774-781.

[7] M. Stanica and H. Guéguen, "Using Timed Automata for the Verification of IEC 61499 Applications," in *Workshop on Discrete Event Systems 2004 (WODES'04)*, Reims, France, 2004, pp. 22-24.

[8] V. Vyatkin, "Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems," in *4th IEEE International Conference on Industrial Informatics (INDIN 2006)*, Singapore, 2006, pp. 874-879.

[9] V. Vyatkin and V. Dubinin, "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499," in *5th IEEE Conference on Industrial Informatics (INDIN 2007)*, Vienna, Austria, 2007, pp. 1183-1188.

[10] J. L. M. Lastra, L. Godinho, A. Lobov, and R. Tuokko, "An IEC 61499 Application Generator for Scan-Based Industrial Controllers," in *3rd IEEE International Conference on Industrial Informatics (INDIN 2005)*, Perth, Australia, 2005, pp. 80-85.

[11] V. Vyatkin, H.-M. Hanisch, and T. Pfeiffer, "Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems," in *1st IEEE Conference on Industrial Informatics (INDIN 2003)*, Banff, Canada, 2003, pp. 224 - 232.

[12] G. Čengić, O. Ljungkrantz, and K. Åkesson, "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime," in *11th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2006)*, Prague, Czechoslovakia 2006, pp. 1269-1276.

[13] K. Åkesson, "Methods and tools in supervisory control theory: Operator aspects, computation efficiency and applications," in *Signals and Systems*. vol. Ph.D. Göteborg, Sweden: Chalmers University of Technology, 2002.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Journal of Software Tools for Technology Transfer,* vol. 1, pp. 134-152, September 1997.

[15] S. Panjaitan and G. Frey, "Combination of UML Modeling and the IEC 61499 Function Block Concept for the Development of Distributed Automation Systems," in *11th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2006)*, 2006, pp. 766-773.

[16] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 5th ed. New York: Springer-Verlag, 2003.

[17] Holobloc Inc., "Function Block Development Kit (FBDK)." (2008, Feburary) [Online]. Available: http://www.holobloc.org.

[18] "Model-Checkers for Net Condition/Event System," (2008, Feburary). [Online]. Available: http://www.ece.auckland.ac.nz/~vyatkin/tools/modelchekers.html.

[19] V. Vyatkin and H.-M. Hanisch, "Modelling and Verification of Execution Control of the Function Blocks following the standard IEC 61499 by means of Net Condition/Event Systems (NCES)," Uni-Magdeburg, IFAT, Technical report March 2000.