V. Dubinin, V. Vyatkin, “Semantics-robust Design Patterns for IEC 61499”, *IEEE Transactions on Industrial Informatics*, 2012, 8(2), pp.279-290

Semantics-Robust Design Patterns for IEC 61499

Victor Dubinin, *non-member*[1] and Valeriy Vyatkin, *Senior Member, IEEE*[2]

***Abstract* - The international standard IEC 61499 for the design of distributed industrial control systems defines an abstract model of function blocks (FB) which allows many different semantic interpretations. As a consequence, in addition, so-called execution models were proposed to specify the execution order of FBs. The variety of models leads to the incompatibility of tools and hinders the portability of automation software. To achieve a degree of execution model independence, in this paper design patterns are suggested that make FB systems-robust to changes of execution semantics. A semantic-robust pattern is defined for a particular source execution model. The patterns themselves are implemented by means of the FB apparatus and therefore are fairly universal. The patterns can be defined and implemented using the FB transformations expressed in terms of Attributed Graph Grammars.**

***Index terms* – IEC 61499, software engineering, semantics, design patterns, refactoring, portability, robustness.**

I. Introduction

The main trend in the development of industrial automation systems is the shift from centralized systems to distributed intelligent systems. This trend was reflected in the development of a new international standard IEC 61499 [1], [2]. The standard supports the design paradigm based on function blocks (*FB*). Despite the undeniable advantages of this concept, it was discovered that the standard has some semantic ambiguities [3]. This may lead to an unacceptable situation when the same FB system will have different behaviour when executed on different platforms. This hinders the portability of automation software developed following the IEC 61499 standard.

To resolve the semantic ambiguities of the standard, several models of FB execution were proposed and implemented. These include: the "non-preemptive multithreaded resource" (*NPMTR-model*) [4], the "interrupted multithreaded resource" (*PMTR-model*), the model based on the sequential hypothesis [5], the cyclic model [6, 7], the synchronous model [8], Petri net based models [9] as well as models implemented in runtimes *μCrons* [10], *FUBER* [11], and *CEC* [12]. In addition, there are models proposed for the implementation of some elements of the standard, namely: for composite FBs [13] and basic FBs [5]. A few efforts have been undertaken to categorize FB execution models on the basis of various criteria. For example, in [3], two criteria to identify so-called “implementation approaches” have been chosen: 1) FB Scan Order and 2) Multitasking Implementation. However, as practice has shown, for the complete specification of FB execution models much more criteria are required [24]. For specification of different FB execution models a special graphical notation *XNet*, has been even proposed [36].

After the semantic problems have been pinpointed in a number of research publications (e.g. [3], [4]), the o3neida community formed a taskforce to resolve them. As a result, the compliance profile [14] has been developed. Based on the recognition of existing practices, that document narrows the variety of semantic interpretations down to three models of FB implementation: sequential, parallel and cyclical. An overview of these models can be found in [16]. However, the portability between these models still remains a problem.

While elimination of the semantic ambiguities in IEC 61499 can be seen as the ultimate solution, in practice it is hard to expect or can take long time to achieve. There are already several tools on the market compliant with the standard, but following different execution models. The method, proposed in this paper can *immediately* help in migrating applications from one tool to another, for example, from ISaGRAF [17], implementing the cyclic execution model, to NxtControl [18], implementing the sequential model, or to the synchronous compiler [8].

To solve the portability problem, this paper proposes *semantics-robust design patterns* (*SRDP*). As illustrated in Figure 1, an SRDP is applied to a function block application (FBA) originally designed to be executed in some *source* model (the *original FBA*), and results in a functionally equivalent application (the *resulting FBA*) that exhibits the same behaviour in one, some, or even an arbitrary *target* execution model.

Conceptually, the application of the SRDPs consists in three steps: 1) adding some new (service) function blocks, 2) changing some function blocks in the original application and 3) changing some of their interconnections.

The function blocks of the original FBA and their counterparts in the resulting FBA will be referred to as *working* FBs, as opposed to the added *service* FBs (not to be confused with Service Interface FBs of IEC 61499). The working FB set is coloured in grey in Figure 1, and the service FBs are coloured in white.

In general, SRDP can be regarded as *specific* software design patterns for *implementation* of projects based on the IEC 61499 standard. It should be noted that currently there are some design patterns for the automation software based on the IEC 61499 standard proposed in [21], in particular: *"Distributed application"*, *"Proxy"* and *"Model-View-*

[1]V. Dubinin is with the Department of Computer Science, University of Penza, Penza, Russia (e-mail: victor_n_dubinin@yahoo.com)
[2]V. Vyatkin is with the Department of Electrical and Computer Engineering, University of Auckland, Auckland 1142, New Zealand (e-mail: v.vyatkin@auckland.ac.nz)

Controller". In contrast to these patterns SRDP is not intended for manual design.

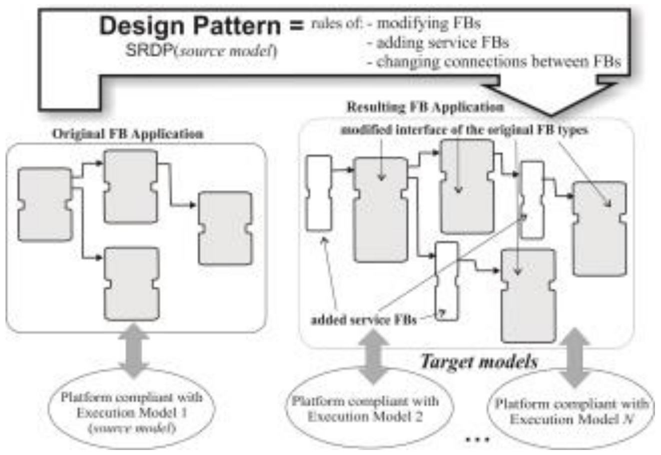

Design Pattern = rules of: - modifying FBs
- adding service FBs
- changing connections between FBs
SRDP(*source model*)
Resulting FB Application
modified interface of the original FB types
Original FB Application
added service FBs
Target models
Platform compliant with Execution Model 1 (*source model*)
Platform compliant with Execution Model 2
Platform compliant with Execution Model *N*
...

Figure 1. General pattern of the Semantic Robust Design Patterns application.

SRDP could be represented as a set of principles, rules, procedures, and partial design solutions. The proposed patterns are implemented by means of the FB apparatus, and therefore are quite universal. The use of the FB apparatus to define the semantics of FB execution models is in some respects similar to defining the semantics of the *UML* by means of a limited subset of *UML* itself [22]. Here also one could draw an analogy with development of an upper ontology belonging to foundational ontologies in the area of conceptual modelling, for example, *Unified Software Modelling Ontology* [23]. In this case a minimal set of concepts and relationships which are necessary for building more specific domain ontologies, are picked out.

Application of *SRDP* can be greatly facilitated by automatic translators. The automatic transformation of the original FBA to the target FBA application can be defined and implemented in many ways, in particular, using the transformations technique based on the use of *Attributed Graph Grammars*, proposed by the authors in [20] for refactoring of FB applications.

In the following Section II the IEC 61499 architecture will be briefly introduced including the main ambiguities of event processing in basic function blocks, whose handling is essential for the proposed patterns. The outline of the proposed solution is presented in Section III. There the structure of the rest of the paper is discussed.

II. Function Blocks Reference Architecture

A. Overview

The artefacts of the IEC 61499 function block architecture relevant to this paper are [1]:

- Function Block (FB) – is a module with interface that consists of event and data inputs and outputs. The events also will be further referred to as *signals*, especially when their processing and transfer are concerned. A function block can be invoked only by an input event.
- The functionality of a Basic Function Block is defined as a state machine called Execution Control Chart (ECC). The semantics of ECC is similar to Moore finite automata with actions assigned to states. An action consists of an algorithm and an output event issuance (either can be omitted). The states in ECC are referred to as EC-states, and the transitions as EC-transitions. An EC-transition has a condition “clocked” by no more than one event input and having a guard condition that is a predicate over data inputs and internal variables (but no events). More precisely, the functioning of a basic FB could be defined by two interacting automata – the control one (that is called Operating State Machine – *OSM* [1]), and the operational one (ECC).
- A Composite Function Block is specified by interface and functionality, defined as a network of function block instances interconnected via event and data connections.
- A Function Block Application (*FBA*) is also a network of function block instances, but it has no interface. An FBA is the structure at the highest level in the hierarchy of IEC 61499 artefacts considered in this paper.
- A service interface function block (*SIFB*) for the purposes of this paper can be understood as a “black box” whose internal structure is not specified.

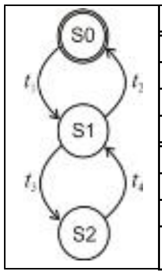
Examples of the introduced artefacts will be encountered by the reader further in the paper. The full list of IEC 61499 artefacts can be found in [1].

B. Ambiguities in Basic FB execution

Processing of input events (in a basic FB) is one of the most ambiguous aspects of the FB execution model. This is due primarily to the fact that the standard does not determine the full lifetime of input events, in particular, it does not define when the events are to be cleared.

Processing of input events by a basic function block is defined in Table 1 of the standard by the operation state machine (*OSM*) (Figure 2). According to this definition, arrival of an input event signal triggers the transition *t1* in OSM. In other words, the interpreter calls the *ECC* which evaluates the *EC* transitions. Finding no enabled transitions, the *OSM* returns to its initial state *s0*. The standard does not say what happens with the input event signal. However, one can assume that if the input event signal is not reset, then the ECC interpreter would be called continuously, which is counterproductive. Thus, one can assume that the "insignificant" input signal is discarded.

As follows from the *OSM*, once activated by an input event, a function block can “jump” through several ECC states before going again to the idle state (*s*0). This activity period of an FB (while OSM is in states *s*1 or *s*2) is referred to as a *single run*.



State		**Operations**
s0		---
s1		Evaluate transitions
s2		Perform actions
Transition	**Condition**	**Operations**
t1	Invoke ECC	Sample inputs
t2	No transition clears	
t3	A transition clears	
t4	Actions completed	

Figure 2. ECC operating state machine (OSM) [1].

In the following typical situations arising at the event inputs of a basic FB, and the ways to address these issues are considered. The "good" situation, unambiguously interpreted by the standard and not causing any problems in FB execution models, is shown in Figure 3. This is the case when one of the event inputs receives a signal and there is one enabled transition in the Execution Control Chart “clocked” by this event.

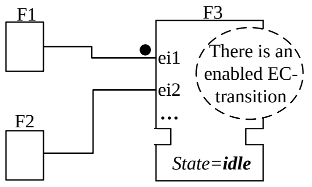


Figure 3. Unambiguous situation with input events.

Alternatively, in the most general case of parallel asynchronous operation of FBs, there can be other situations which may be interpreted ambiguously. Let us call these situations *problematic*. Schematically all possible problematic situations are shown in Figure 4. In this illustration, the state of the *OSM* of the resource executing this FB network is designated as *State*. The value *idle* corresponds to the *OSM* state *s0*, and the value *busy* corresponds to the states *s1* and *s2*.

In parallel operation, it is possible that event inputs of an FB receive several signals simultaneously, as illustrated in Figure 4(a). Another possible situation, when a signal arrives in the busy state of the FB, is shown in Figure 4 (b). This situation is more likely to occur the longer is the execution duration of algorithms in the ECC. The third possible scenario is shown in Figure 4(c). Here the event is “not expected” in the current ECC state, i.e. no EC-transitions from this state are “clocked” with this event input variable. Simultaneous arrival of several events in the problematic situations (b) and (c) would not bring any essential differences, so one can omit them. It should be noted, however, that not all execution models allow the existence of these situations. For example, in those sequential models where execution of an FB is atomic, the situation (b) is impossible. Situation (a) is impossible in the NPMTR model [3] of FBRT and in the execution model based on the sequential hypothesis [5] but is common for the synchronous and cyclic execution models.

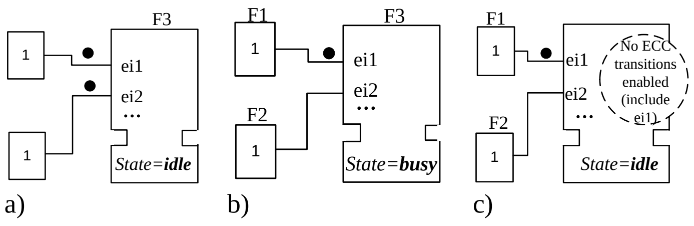


Figure 4. Problematic situations with input events: a) Simultaneous arrival of signals to the inputs of FB; b) Arrival of the signal at a busy FB; c) Arrival of the signal at idle FB, when its processing is not envisaged in the ECC.

To ensure the determinism of FB application behaviour, an execution model must unambiguously resolve the problematic situations. There are following options for handling input events in the situation (a):

1) Using some rule (e.g. a priority, pre-defined or based on the time of arrival), select one input event signal for processing, and discard the others. This interpretation is more consistent with hardware implementation of function blocks (e.g. [15]), when the signal is understood as a pulse and there is no explicit buffering of signals. The drawback of this approach is a possible loss of the event signal which can be carrying important information;

2) Process all input events, but one by one. The order of processing again can be based on a priority mechanism. The phase of FB activity includes the processing of all input signals sequentially, one after another;

3) One input event is processed, chosen by a certain rule, and the others are remembered for future processing. After processing of this event, one of the other "ready" FBs can be activated. If there are no such FBs, then the next input event signal in the FB is selected and processed. The phase of FB activity includes the processing of one signal.

There are two options to resolve the scenario (b):

1) discard the input event signal; or 2) remember the input event signal and process it later when the function block becomes idle. One can assume that the standard is more inclined towards option 1, since its early draft was using the (discarded) option 2 which used signal queues of length 1 (so-called EI-variables) (see [26]).

The same two options exist for the scenario (c): the signal at the event input can be discarded or not. The standard proposes the first option by an implicit rule: "If in some *EC-state* an input event signal arrives that is not included in any transition from this *EC-state*, then the input event signal is discarded."

The following assumptions concerning the processing of input event signals are made when discussing the implementation patterns:

1) Scenario (a) is resolved by using buffers;

2) Scenario (b) is excluded from the consideration since the source execution model assumes FB execution to be atomic;

3) Situation (c) is resolved following the first option (event signal is discarded).

It is also assumed that in all execution models the syntax of basic function blocks, in particular of ECC, prescribes EC transition to be clocked with one or zero input events, and does not allow using event names in guard conditions of transitions.

III. Overview of the Proposed Method

A function block application A_S designed for a known execution model *S* (source) will be transformed to an application A_U = *SRDP* (A_S, *S*) that will have equivalent behaviour in any[2] other (unknown prior to the transformation) execution model (The U in the index stands for "universal"). The transformation consists of modifications of the function block types used in A_S, along with adding, in some cases, a *global scheduler* function block. The transformations of FB types are as follows:

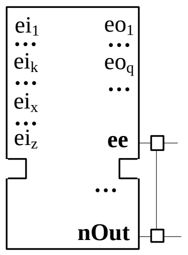


Figure 5. Changes in the working basic FB interface for the use in SRDP.

- For basic FB types a generic transformation method is proposed (i.e. independent from the source execution model *S*) which enables determining the FB execution termination at any input events arrived in each state at any input data variable's values;
- For composite FB types, the transformation method is specific to a particular source model. Dependent on that, the transformation may include insertion of the following service FBs:
  - Buffers of event signals for each working FB ensuring delivery of all intended event signals as well as the desired order of input event signals' processing;
  - Counters of events indicating the reception of input events by the buffers. These are needed to determine correctly termination of the buffer's work in the signal recording synchronization system ;
  - Event transmission controllers responsible for moving events between the buffers;
  - Local schedulers, ensuring that component function blocks with this composite FB are invoked in the same order as in the source model;

Altogether, these transformations aim at ensuring the equivalence of resulting behaviour to the original one in the source model.

It should be noted that if the target model of execution *T* is known, then it is possible to develop a particular pattern SRDP'(A_S, *S*, *T*) that will create a more efficient A_T equivalent to A_S but only when executed on the target platform *T*. Note the difference of arguments between SRDP' and SRDP.

The rest of this paper is structured as follows. Section IV introduces the transformation rules to be applied to basic FBs in order to achieve their event input order robustness. Section V presents buffering of signals that is another key enabling mechanism of SRDP. Section VI considers SRDPs application to composite FBs on example of cyclic and synchronous execution models. Section VII presents a comprehensive example of SRDP application. The paper is concluded with Conclusion and References.

[2] of course, within the defined set of execution models.

IV. Transformations of Basic Function Blocks

A. *Interface transformation*

The goal of the proposed basic FB transformations is to make them signalling the termination of ECC execution by emission of an (added) event output *ee*, the same time outputting the number of events emitted by the FB during this *single run* using an added output data variable *nOut*. This is needed to determine the end of the signals' transmission from the working FB to the appropriate buffer. It should be noted that if the target execution model uses the single-stage transmission of signals between FBs (e.g. synchronous or cyclic execution models), then there is no need for the variable *nOut* and the completion of the signals' writing to the buffer has to be determined by the buffer itself (without using variable nOut). The modification of the FB interface is illustrated in Figure 5.

B. *ECC transformation*

ECC of the FB needs to be modified accordingly in order to achieve this interface behaviour.

The termination signal (*ee*) must be emitted even in case of the so-called *insignificant input event signal*, i.e. when the input event triggers no transitions in the ECC. Indeed, *ee* should signal the *OSM's* transition to the state *s0* (transition *t2* in Figure 2). The issuance of the *ee* signal is accompanied by an update of the output variable *nOut*. If no output event is issued, then its value is zero.

The ECC transformation rules will be presented using the notation of Attributed Graph Grammars. The left side of the rule is a graph structure that is to be substituted by the graph in the right side. The notation of [20] will be followed, which terms an arc with only a guard condition as *C-arc*, *EC*-transition having event input name in the condition as *E-arc*, and a transition with a constant guard condition equal to *true* as *T-arc*.

The ECC transformation will be done in four steps. First, an equivalent transformation (refactoring) will be applied to eliminate the states where both *C* and *E* arcs originate. As a result, the set of EC states will be divided onto two sets: *terminal states*, where only *E* arcs originate, and transitional states, where only *C* arcs originate. This is possible as shown in [20]. Second, the *ee* signal emission will be added to all terminal states. The third transformation is applied to the terminal states to ensure that *ee* signal will be emitted for all input events signals and at any data variables' values. The fourth transformation's aim is to count the number of output signals issued by the FB. For deeper understanding of rules semantics one could present their left and right hand sides in the form of *XNet* [36]. However, in the given case ECC execution model to be considered is constant (and, it is in general use, see [20]), therefore the rules stated below are quite obvious from the logical point of view, without additional semantic expositions.

1) *Separation of terminal and transitional states*

An *EC*-state is called *terminal* if it has only outgoing *E-arcs*. When transitioned to a terminal *EC*-state, the FB completes its execution and waits for receipt of new events. Thus, signal *ee* needs to be issued only in terminal *EC*-states.

The transformation rule in Figure 6, which complements the rule set from [20], eliminates *EC*-states s_i where both *E*-arcs and *C*-arcs originate. For that, one introduces an additional state s_a and the arc (s_i, s_a) which has the guard condition $\sim c_k \& \ldots \& \sim c_n$ defining an explicit transition from *EC*-state s_i when all the guard conditions on the outgoing *C*-arcs evaluate to *FALSE*. The generated s_a state is terminal.

2) Add ee emission to terminal states

Then the emission of output signal *ee* is assigned to all terminal states. Now, the *ee* signal will be emitted at the end of each run. Figure 6 shows the combined application of both transformations (1) and (2).

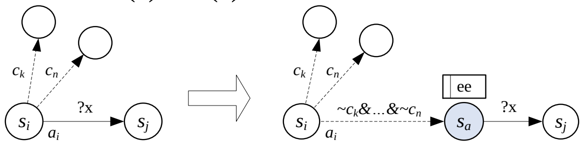


Figure 6. Rule for determination of the terminal EC-state in the case of outgoing both *E*- and *C*-arcs.

3) Emit ee signal for all input events and all input data

However, the transformed ECC would not emit the *ee* signals if the function block receives an *insignificant* event input not causing any state transition (this may be because this event is not a part of any transition condition originating in the current state, or if no guard conditions of such a transition evaluate to *true*). The proposed transformation solves this problem by modifying the outgoing arcs of terminal *EC*- states such that an EC transition occurs even at an insignificant event arrival. The generalized transformation rule is shown in Figure 7. The following notation is used here: symbols *ei*, *eo*, *a* with indices stand for event inputs, event outputs and algorithms, respectively. The guard conditions associated with event input ei_k are denoted as c_{k1}, c_{k2}, ..., c_{kn}.

The state s_i is supplemented by the “loops” implementing the s_i state preservation. The number of such “loops” is equal to the number of event inputs in the FB.

Each loop consists of a transition to an auxiliary state (marked as filled circles) and transition back to s_i (the latter is with *true* condition). There are two kinds of conditions on the arcs of the loops:

(1) for all event input signals $ei_g (g = \overline{1,k})$ that clock the outgoing arcs of s_i, the condition on the corresponding arc is formed as a conjunction of negated guard conditions of all the outgoing arcs of s_i that include ei_g. The number of cycles of this kind is the number of different input signals, marking the outgoing arcs of s_i (ei_1, …, ei_k).

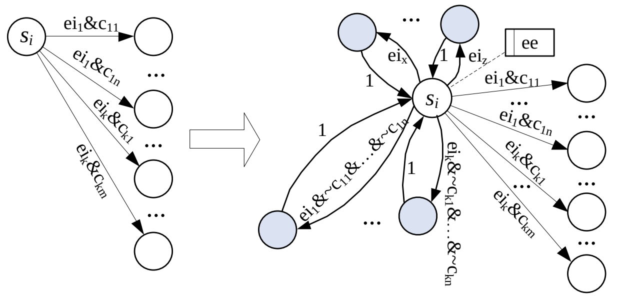


Figure 7. Rule for enforcing transitions from terminal *EC*-states.

(2) for all other event inputs of the FB, the conditions are formed just from the corresponding event input signals (ei_x,…, ei_z in Figure 7, right part).

4) Count the number of output signals issued by FB

At first glance, this task is trivially solved by complementing each algorithm of an *EC*-action in which an output signal is issued, by a statement incrementing the counter *nOut*. It can be achieved by introducing a new special *EC*-action A_I doing this function. However, the situation is complicated by the possible emergence of loops in terminal *EC*-states that arise from application of the rule (3). The rule in Figure 8 solves this problem by moving the *EC*-actions from an “inconvenient” terminal *EC*-state s_j to an intermediate state s_a and extending the *EC*-actions by incrementing *nOut*. It is necessary because events on the loops in state s_j could invoke the same actions, but would cause incorrect behaviour.

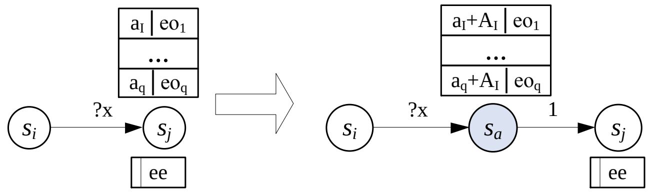


Figure 8. Rule for moving EC-actions from inconvenient *EC*-state and adding incrementing *nOut*.

The reset of the *nOut* counter could be done in the first statement of an algorithm of the first *EC*-action attached to the target *EC*-state of an *E*-arc outgoing from the terminal *EC*-state. It can also be achieved by introducing a new special *EC*-action A_0 doing this function. The rule adding A_0 to an *EC*-action for resetting the counter *nOut* is shown in Figure 9.

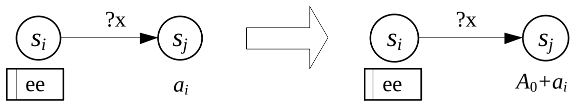


Figure 9. Rule for bringing in *EC*-action A_0 for resetting the counter *nOut*.

The rules should be applied to an ECC in the suggested order: from the rule in Figure 6 to the one in Figure 9.

The presented *ECC* transformation preserves the original behaviour but adds the emission of *ee* signal and returns the number of emitted events in *nOut*. The equivalence can be rigorously proven, but the proof is omitted due to the space constraints and the intuitive clearness of the transformations properties.

C. Implementation example

The proposed transformation is an evolution of the *ECC* refactoring technique proposed in [20]. For each of the rules in Figures 6-9 in the general case a few rules of graph transformation system *AGG* [28] have been developed, which were added to the set of rules of the existing *ECC* refactoring system. The resulting rule base has been tested in *AGG* on several examples.

In this paper, the transformation of *CruiseController* basic FB is considered. This FB is the main part of the cruise control system presented in [8]. For better readability of the figures, a brief notation for elements of *ECC* presented in [8] is introduced. Signals e_0, e_1, and e_2 will represent the signals *INIT*, *SpeedChange*, and *DesiredSpeedChange*, respectively. The names of event outputs (same as of the *EC*-actions) *INITO*, *ThrottleOff*, *ThrottleUp*, and *ThrottleDown* are abbreviated to a_0, a_1, a_2, and a_3, respectively. Also, the following notation for guard conditions is used:

$c_1 \equiv CurrentSpeed=DesiredSpeed;$

$c_2 \equiv CurrentSpeed<DesiredSpeed;$

$c_3 \equiv CurrentSpeed>DesiredSpeed;$

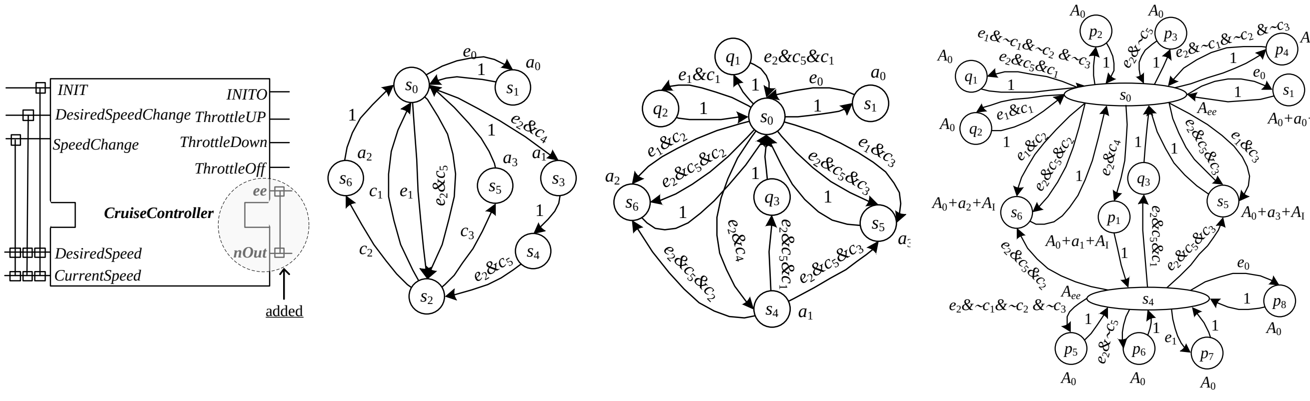


Figure 10. Transformed interface of the CruiseController FB (a), original ECC (b), refactored (c) and transformed ECC (d).

$c_4 \equiv DesiredSpeed=-1;$ $c_5 \equiv DesiredSpeed>-1.$

The interface transformation is presented in Figure 10 (a), the original *ECC* in Figure 10(b), the refactored *ECC* in accordance with [20] in Figure 10(c) and the resulting *ECC* in Figure 10(d).

In the processing of such a transformation some *EC*-state can be deleted (for example, *EC*-state s_2 has been deleted during the refactoring) and some new *EC*-state can be created (these EC-states have been named using identifiers *p* and *q* with indexes). In Figure 10(d), A_{ee} denotes an *EC*-action issuing the *ee* signal.

V. Buffering Event Signals

Buffering of event signals is required in the implementation of several execution models, for example in sequential model implemented in FORTE runtime, as discussed by Zoitl in [19]. Therefore, along with the introduced basic FB modification, buffering of event signals is another important implementation mechanism of SRDP. The controlled buffer determines the rules of transferring, buffering, and processing input event signals. The choice and the implementation of such a buffer are determined by the source FB execution model, namely by properties of buffering which need to be preserved after transferring an application to the target execution model. It should be noted that some execution semantics originally may not require buffers, but maintaining the same event scheduling mechanism on the target platforms will require it. An example can be reproducing synchronous model rules in the sequential semantics.

It is possible to implement buffers supporting various disciplines of buffering (e.g., *FIFO*, *LIFO*, priority-based one) as well as memorization of multiple signals for the organization of queues of events at event inputs of FB. However, in the following the *FIFO*-buffer with preservation of only one event for one event input will be used, because it is most consistent with the standard IEC 61499, and most FB execution models as well.

The interface of the *Buffer* FB is shown in Figure 11(left hand side). The inputs ei_1, ..., ei_n correspond to input event lines, through which the signals arrive to be placed in the buffer. The event output lines eo_1, ..., eo_n are used to output the corresponding signals from the buffer. Recording of the event input event signal ei_k to the buffer is confirmed by the corresponding output event signal ack_k. Exactly one signal is output when the buffer receives the *putOut* input event signal. If the buffer is empty, then the *empty* output event signal is emitted.

The ring buffer can be efficiently implemented (as a SIFB) using three simple and fast operations: increment of the index, direct writing (reading) the identifier of event input to (from) the buffer, and comparison with the upper bound of the array. The time-sequence diagrams describing the work of the buffer are shown in Figure 11 (right hand side).

To estimate the complexity added by the use of buffers, one can assume that the durations of these operations are the same and equal to t_1. Because each signal is first written and then read from the buffer, the delay in the transmission of each signal between a pair of FB (provided that only one buffer is between them) is $6 \times t_1$ while the transfer of *n* signals needs $6 \times n \times t_1$.

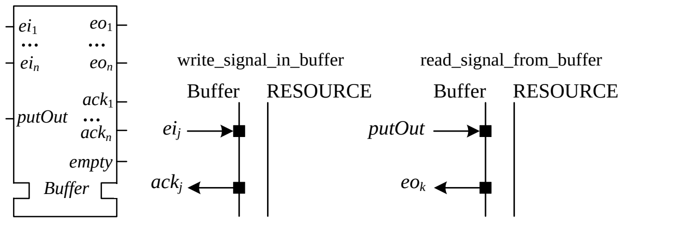


Figure 11. Buffer FB interface (left side) and time-sequence diagrams describing the work of the buffer (right side).

As it will be seen in the following sections, there are two typical structures of buffer interconnections arise when SRDPs are applied to FB networks. These are referred to as *inb* and *outb*, as conceptually illustrated in Figure 12. For example, in the *inb* structure all events from the input buffer buf_i are sequentially processed by the block fb_i and written to the output buffers buf_j, ..., buf_k.

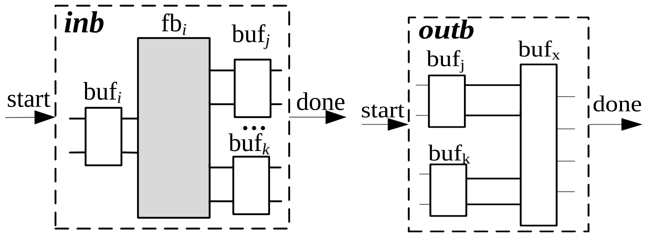


Figure 12. *inb* and *outb* - structures of the events flow.

VI. Transformation of Composite Function Blocks

In this Section some concepts of SRDP application in composite FBs will be discussed using as examples the cyclic and the synchronous models of execution.

A. Cyclic execution model

There are several models of composite function blocks execution, namely: as a single entity [13], and as a container [26]. In this paper the second model is assumed. As it was shown in [26], a hierarchical FBA built from composite and basic FBs, can be reduced to a flat FBA containing only basic FBs and SIFBs. The process of “flattening” includes the insertion of data valves - the intermediate storage elements implementing the interface logic. The data valves could be efficiently implemented using SIFBs.

The cyclic execution model implies that the order of FB execution in a system is explicitly assigned. The global execution order can be unambiguously determined from the local orders within each composite FB. This is illustrated in Figure (a) on example of some abstract hierarchical FB application, where A, B, D, and E are composite FB types, and C, F, G, H, I and J are basic FB types. The local execution order within a composite FB is defined by the numbers assigned to the blocks. The global order of execution can be defined as follows: *G, H, D_1", D_2", E’, I, J, E", F, B", C*. Here, the symbol with a prime denotes the activity of input interface logic of the composite FB, a symbol with a double prime - the activity of output interface logic. Function block *D* has two event outputs, therefore two separate output data valves D_1" and D_2" are distinguished.

Since the activity of a composite FB is composed mainly of the activities of its component basic FBs, one has to focus upon basic FBs. In accordance with the given local order, the global execution order of basic FBs in the flattened systems will be: *G, H, I, J, C*.

Figure (b) shows the one-level (flat) representation of the same FBA. The appeared data valves are denoted as *dv1-dv6*. The numbers under the FBs and the data valves denote the

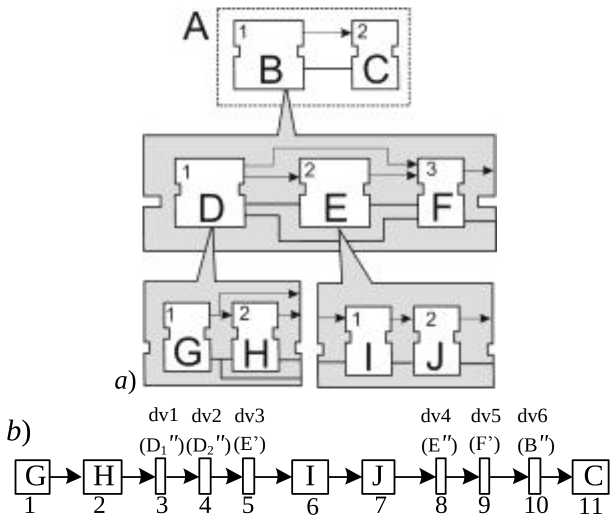


Figure 13. Hierarchical FB structure with local orders of execution (a) and its flattened representation with global order assigned to FBs and data valves (b).

global order of their execution. As seen from this figure, the data valves are scheduled uniformly with the rest of the FBs in the application. This flat structure can be used to achieve the behaviour equivalent to the cyclic in any execution model since it preserves the execution order of the original application.

Another approach to SRDP of composite FB in the cyclic source model is to use "native" interface logic of the composite FB without flattening the entire application, but it will not be presented due to space limitations. It has many similarities with the synchronous model implementation considered in the next subsection.

B. Synchronous execution model

The synchronous execution model [8] belongs to the class of parallel execution models. This means that several FBs can be active simultaneously. There are several varieties of synchronous execution model depending on the adopted granularity [16], e.g. based on single ECC transition, or single FB run.

However, the corresponding target application will be executed in a sequential way. Therefore, the pattern discussed in this section shall emulate the synchrony and parallelism of the original application.

Synchronous models can be classified onto single-stage and two-stage models. The two-stage FB implementation scheme with intermediate buffering of output signals [35] is used to avoid the dependency of the execution result on the order of invocations of the FBs belonging to the set of to-be activated blocks. At the first stage, all the FBs which are ready to run are invoked, but the transmission of output event signals from the FB-producers to the FB-consumers is delayed. One can think that these signals are "frozen" for a moment, being stored in some intermediate buffer. At the second stage the delayed signals are delivered simultaneously to the consumer FBs. Execution of all active FBs is synchronized with a single pulse (*tick*).

The main problem in implementation of composite function blocks within the "synchronous execution model" pattern is to implement the second phase, consisting of the transmission of signals from the output buffers to the input buffers when the source- and receiver-buffers are in different ambient FBs belonging to different levels (or different branches) of the system's hierarchy. In the first phase this problem does not occur because all its participants (the input buffer, the FB and the output buffer) are localized within the same ambient composite FB.

The aforementioned problem is illustrated in Figure 14 for the application composed of three function blocks fb_1, fb_2, and fb_3. Here one observes the *outb* structure of buffers from Figure 12 for the case where all buffers belong to *sibling* FBs in the system's hierarchy. For simplicity, in this figure the exact event flow links between the buffers is omitted by showing the directions of event flow by wide arrows. In addition to the actual buffers, the structure includes the signal transmission controller d_x of type *dispOutMoving* that implements moving of event signals between output buffers buf_j and buf_k of FBs fb_a and fb_c, respectively, and input buffer buf_x of FB fb_x. As seen from Figure 14, as a result of the *outb* "decentralization", several event links are broken at the FB borders. To connect them, new event inputs/outputs are added to the interface of the corresponding FB.

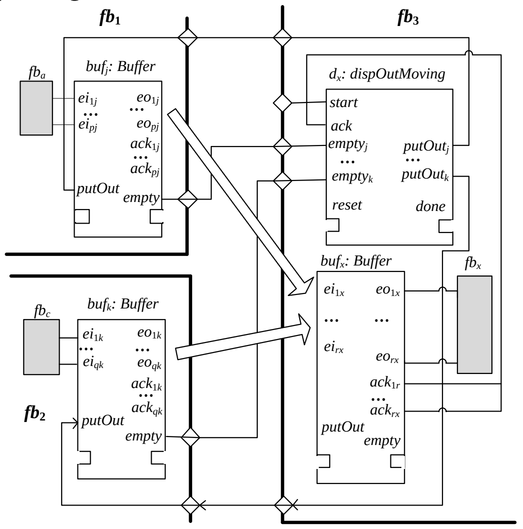


Figure 14. Detailed structure of decentralized *outb*, when all the buffers are located in different ambient FBs.

The transformation of the composite FB's contents will depend on whether the component FB is basic or composite. In the following the rules governing the implementation of composite function blocks within the synchronous execution model are presented:

1) For a basic component FBs, one input and output buffer of signals are created, as well as a counter of the acknowledgements from the output buffers;

2) For a composite component FB, no additional (infrastructural) block is created;

3) The controller of signals' transfer of type *dispOutMoving* is placed (if necessary) in the composite FB which hosts the receiving input buffer;

4) The composite FBs of intermediate level are appended by a pair of FBs of types *gather1* and *gather2*, whose function is to gather acknowledgements from the sub-systems of types *inb* and *outb* about the completion of their execution and the formation of the output signals indicating the completion of the first and second synchronous execution phases in this composite FB. The FB types *gather1* and *gather2* can be regarded as schedulers, whose function is reduced to managing a single execution of one phase in the local area;

5) Similarly to the cyclic model case, the main scheduler is added only once at the resource level (i.e. to the FBA of the top hierarchy level).

The working of each phase (first and second) in the whole system consists of the working on the FB implementation conducted in local areas (i.e. in composite FBs). The hierarchically interconnected system of blocks *gather*1 (rsp. *gather2*) forms a system for determining the termination of the sub-systems *inb* (*outb*) on a global scale.

Figure 15 shows conceptually an example of converting an intermediate level composite FB containing composite (*D* and *F*) and basic (*F*) component FBs. The group corresponding to the basic FB (*F*) is surrounded by a dashed line. For simplicity, it was assumed that the basic FB *F* has only one basic FB as its event "predecessor" so the controller of signals' transfer is not required.

Although the resulting FB is obviously more complex than the original one, the complexity added by the service FBs intuitively is of the same order of magnitude as the original application.

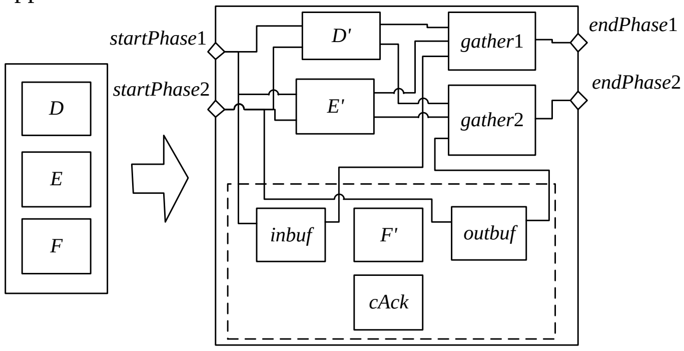


Figure 15. An example of converting an intermediate level composite FB containing composite and basic component FBs, in the synchronous execution model.

VII. Example Application

In this section the use of the SRDP is illustrated on example of a composite function block implementing a simple cruise control in a car, slightly modified from the one presented in [8]. As seen in Figure 16, it consists of four function blocks, each modelling a component of the system.

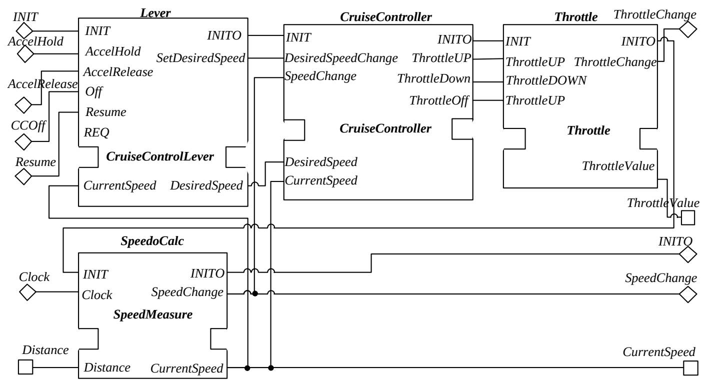


Figure 16. Composite function block *CruiseControl* designed to be executed in cyclic execution model.

The cruise control system is activated by a lever, consisting of the *Accel* and *Off* buttons. Whenever the *Accel* button is held down, a sequence of *AccelHold* events will be generated to incrementally accelerate the car. When the button is released, the *AccelRelease* event will be generated and the speed at that instant is memorized. This desired speed will be forwarded to the cruise controller, which in turn will attempt to maintain this speed by appropriately adjusting the throttle position. A separate subsystem calculates the current speed at every Clock tick and updates the cruise controller. This cruising mode will be deactivated when the *Off* button on the lever is pressed. For the sake of brevity, the ECCs are not presented, the interested reader is referred to [8].

The *CruiseControl* composite function block was designed in the cyclic execution semantics. Applying the implementation pattern one can create an equivalent composite FB which will run correctly in any other runtime environment, originally not supporting the cyclic semantics.

The result of applying the "Cyclic" *SRDP* is presented in Figure 17. The resulting FB is rather complex due to the fact that the complete pattern has been applied to ensure equivalent behaviour of the transformation result in any execution model, including those where the "single run" concept won't hold (like in hardware implementation of FBs or *NPMTR*, e.g. *FBDK*). This is achieved by using the *CountAck* function blocks to determine termination of the buffers' execution. For most other execution models these FBs can be omitted (the *ee* output of the working FB will signal correctly its termination and activate the buffer) thus substantially simplifying the resulting design.

The following signals have been added during the transformation:

startScan – begin the scan of FBs in the order defined in *exList*.

scanDone – the scan is completed.

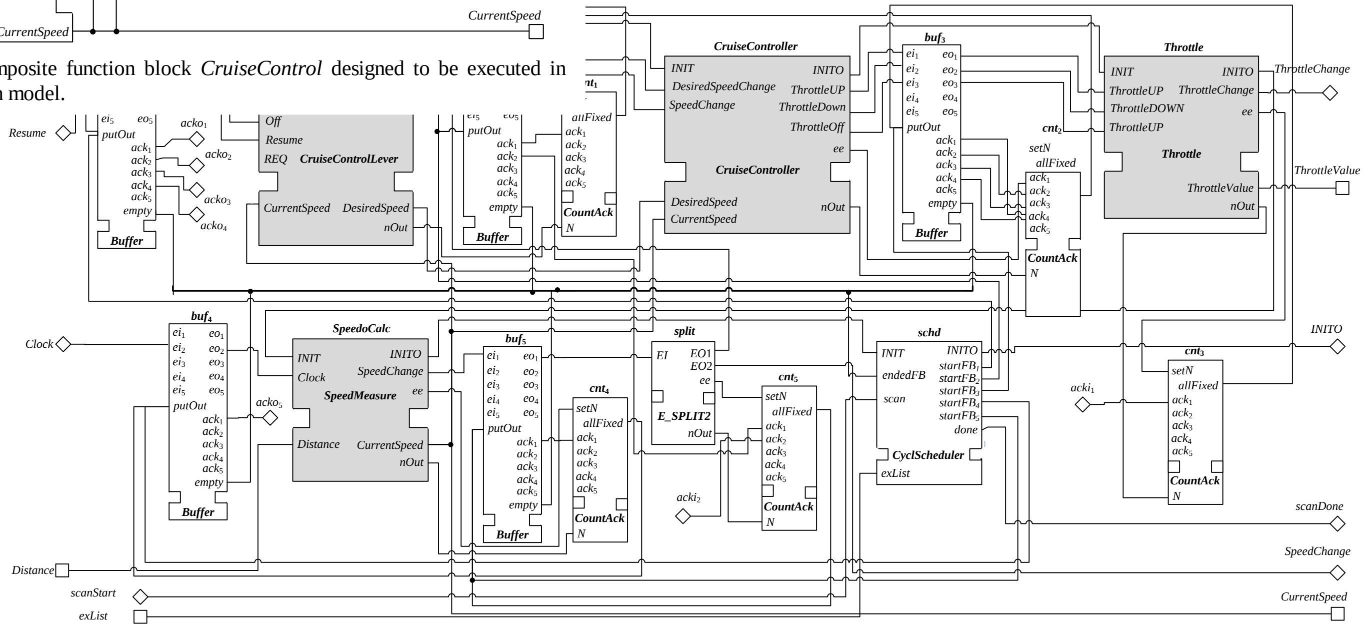


Figure 17. The function block "CruiseController" constructed in accordance with the "cyclic execution model" pattern.

$acki_i$ – acknowledgement of writing to an external buffer situated outside of this FB network;

$acko_j$ – output signal acknowledging writing to a local buffer sent to external *CountAck*.

To implement correctly the signal "forking" from event output *SpeedChange* of *SpeedoCalc* FB, an explicit *E_SPLIT* function block is needed. This FB type also needed to be transformed according to the SRDP. The result, called *E_SPLIT2* is used in the resulting FB diagram.

Let us consider, how this function block network will be executed, say, in *FBDK*, which implements event passing through the direct method call, i.e. sequence of FB invocations can be described by the depth first algorithm of graph traversal. The depth-first event propagation will be stopped in the first buffer causing the FB interpreter to backtrack to the FB- origin of the event and let it emit the next event. Thus, sequentially, one by one, all the signals issued by the working FB, will be written to the buffer until the basic FB transitions to a terminal *EC*-state. Thus, the depth-first traversal will be converted to the breadth-first one which fits to the known FB execution semantics.

VIII. Complexity Estimations

The complexity of the FB system resulting from SRDP application essentially depends on the particular execution model, as well as on the number of system's hierarchy levels and FB network topology.

The degree of complexity of SRDP application can be represented by the formula:

$$K = N_{res} / N_{src} = (N_{src} + N_{add}) / N_{src} = 1 + N_{add} / N_{src} \qquad (1)$$

where: N_{src} – the number of FBs in the original (source) FB system; N_{res} – the number of function blocks in the resulting (target) system; N_{add} – the number of additional FBs appeared in the target system as a result of SRDP application.

A. Evaluation of complexity overhead for cyclic SRDP

The complexity overhead of SRDP shows itself in the increased complexity of basic FB ECCs (which is quadratic to the number of inputs and outputs in worst case) and in the number of added service FBs to composite FB networks, which is linear to the number of FBs in the original network.

Here the complexity of the target FB system for the cyclic design pattern will be estimated under the following restrictions: considered is a flat one-tier connected network of FBs without external inputs-outputs. Let M denote the number of generalized event connections between the FBs: it is said that there is a generalized event-connection between FBs fb_i and fb_j, if there is at least one event connection between these FBs. The number of additional FBs in the system resulting from SRDP can be defined as $N_{add}= N_{buf} + N_{ack}$, where $N_{buf} = N_{src}$ – the number of FB's input buffers, $N_{ack}= M$ - the number of acknowledgements counters.

It should be noted that if a generalized event-link consist of only one ordinary event connection, it can be excluded from consideration because it does not require the use of an acknowledgement counter.

The overhead ratio for cyclic SRDP can be found as follows:

$$K_{cyc}= 1+N_{add}/N_{src} = 1+(N_{src}+M)/N_{src}= 2 +M/N_{src}; \qquad (2)$$

As can be seen from (2), the resulting system contains at least twice as many FBs as the original system.

In the worst case - when the FB network topology is a complete graph $M= N_{src} (N_{src}-1)$, the ratio is:

$$K'_{cyc} = 1+N_{add}/N_{src} = 1+ (N_{src}+M)/N_{src} = 1 + N_{src} \approx N_{src} \quad (3)$$

In this case, the complexity increases proportionally to the number of function blocks in the original system. However, it should be noted that in practice the complete graph topology is unlikely to see in practice.

B. Overheads of synchronous SRDP

The complexity overhead of the synchronous SRDP will be evaluated under the same restrictions as in the previous case. Let L denote the number of FBs with more than one generalized incoming event connections. Then

$$N_{add}= N'_{buf} + N''_{buf} +N_{ack} + N_{mov},$$

where: $N'_{buf} = N_{src}$ – the number of input buffer FBs,

$N_{buf} = M$ – the number of output buffer FBs,

$N_{ack} =M$ – the number of acknowledgement counters,

$N_{mov} =L$ – the number of control units to moving signals from the output buffers to the input ones.

The overhead ratio is:

$$K_{sync}=1+N_{add}/N_{src}=1+(N_{src}+2M +L)/N_{src}=2+(2M+L)/N_{src} \quad (4)$$

As can be seen from (4), the resulting system again contains at least twice as many FBs as the original system.

Let us consider the ratio of the complexity factors for synchronous and cyclic models:

$$k_{21}= K_{sync} / K_{cyc} = (2+(2M+L)/N_{src})/(2 +M/N_{src}) \approx (2M+L)/M = 2+L/M, \text{ when } M/N_{src} >>2. \qquad (5)$$

As follows from (5), the complexity of synchronous SRDP is at least double of cyclic SRDP (Note: this is true if the number of generalized event connections in the original system greatly exceeds the number of FBs).

A more comprehensive estimation of the SRDP impact on the performance will be the subject of future work. For that, new analysis techniques, such as static timing analysis [31] can be applied. Intuitively, the execution complexity added by *SRDP* is the greater the more different are the source and target execution models. Such overheads are always observed when one system is emulated by means of another.

IX. Conclusions and Outlook

This paper proposed semantic robust design patterns of function blocks for three execution models. The goal of the implementation patterns is to increase portability of software built according to the *IEC* 61499 standard.

There are no fundamental restrictions of SRDP, since the expressive power of SRDP mainly depends on the set of service FBs used. However, the ones developed so far and presented in the paper do have a number of restrictions.

First, the considered *SRDP* are focused on such source execution models that are based on the "*Single Run*" principle, that is, the execution granule is a function block. Second, the order of signals' transmission between FBs as well as the order of input event signals processing are largely determined by the buffer types, so when using *FIFO*-buffer, the priority order of signals transmission between FB of the source execution model cannot be kept in the target execution model which do not support prioritized transmissions. Third, the chronological order of output signals issuance (such as in the execution model based on sequential hypothesis) by using ordinary buffering proposed in the paper cannot be preserved in the target execution model. Therefore it requires different approaches to the construction of the system that were considered in the paper as well. Fourth, an execution model which is not based on the *Single Run* principle (for example, *NPMTR*) cannot be used as a source execution model.

A possible downside of the proposed approach is a more complicated resulting code having potentially worse execution performance. However, the performance overhead doesn't seem to be substantial though. The reaction time of each basic function block would not change considerably after the transformation described in Section V is applied. Indeed, the added transition arcs to the terminal states can be made of lower priority than the original ones, therefore for significant events nothing would change. To optimize event scheduling in the cyclic model, the *hot-potato principle* [30] can be used to transfer generated signals across the interfaces as soon as they are emitted.

The power of *SRDP* is also restricted by the *computational power* of the FB model of computation. In the standard IEC 61499 the FB model is implicitly defined as a parallel and asynchronous (that caused multiple interpretations). To the best of the authors' knowledge, no specific research was dedicated to define the computational power of the FB model, but on the basis of existing publications [11, 9, 32], one can assume that the computational power of the FB model (even without the use of complex algorithms) is not less than that of the *Safe Free-Choice Petri Net* (subclass of Petri nets [33]). Using algorithms in basic FB and, moreover, various services in *SIFB* makes the FB model (w.r.t. to the computational power) equivalent to the universal algorithmic systems. Therefore, there are should not be any theoretical restrictions on the use of *SRDP*.

To prove rigorously the equivalence of FB systems' behaviour on source and target platform, one needs to specify what the equivalent behaviour is, which may include the preservation of several orders: the FB execution order, the order of the output signals issuance to the environment, the order of signals' transmission between the FBs, etc. Then one needs to develop formal models of FB systems behavior and compare the corresponding prefix languages generated by both models. The mathematical apparatus developed in [26] can be used for that. Another approach can be based on the use of *model checking*, in which the properties of FB system are formulated based on temporal logic. This approach focuses on equivalence of states rather than languages. However, the proof of the equivalence of formal FB models is a standalone research topic deserving a separate investigation.

Another direction for further work is software implementation of the converters, transforming the original FB application to a semantically equivalent FBA in a selected tool, for example, run-time environments *FORTE* [10], *ISaGRAF* [17] and Synchronous Compiler [8]) and in the modified IEC 61499 execution semantics as per the second edition of the standard). The authors have already implemented a prototype converter using the Attributed Graph Grammars for ECC and in C++ for composite FBs. The ECC transformation in section IV has been done using the AGG tool. Furthermore, it has been shown in [34] on example of transformation of IEC 61499 applications to *Net/Condition Event Systems* (an extension of Petri nets) that there are no fundamental obstacles to apply graph transformation to the entire IEC 61499 projects.

Acknowledgements

This work is supported, in part, by the FRFD grant 3625072/9573 of the University of Auckland.

X. References

1. Function Blocks for Industrial-Process Measurement and Control Systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
2. V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State of the Art Review", *IEEE Transactions on Industrial Informatics*, 7(4), November, 2011
3. L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in Proc. 2nd *IEEE Intl. Conf. Industrial Informatics* INDIN'04, Berlin, Germany, 2004.
4. C. Sünder *et al*, "Usability and Interoperability of IEC 61499 based distributed automation systems", 4th IEEE Conference on Industrial Informatics (INDIN '06), Proceedings, Singapore, 2006
5. Vyatkin, V., Dubinin, V. Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499, Proc. 5th IEEE Intl Conference on Industrial Informatics (INDIN'07), July 23-26, 2007. – Vienna, Austria, 2007. - pp. 1137-1142.
6. J. LM Lastra et al , "An IEC 61499 Application Generator for Scan-Based Industrial Controllers", in Proc. of the 3rd IEEE Conference on Industrial Informatics, Proceedings, Perth, Australia, August 2005.
7. P. Tata, V. Vyatkin, "Proposing a novel IEC61499 Runtime Framework implementing the Cyclic Execution Semantics", 7th IEEE Intl Conference on Industrial Informatics (INDIN'09), Cardiff, Wales, June, 2009
8. L.H. Yoong *et al*, "A Synchronous Approach for IEC 61499 Function Block Implementation," IEEE Trans Computers, Vol. 58, Issue 12, 2009
9. Hagge, N.; Wagner, B.;"A new function block modeling language based on Petri nets for automatic code generation," *Industrial Informatics, IEEE Transactions on* , vol.1, no.4, pp. 226- 237, Nov. 2005
10. PROFACTOR Produktionsforschungs GmbH, "4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment," 30/10/2007 2007; http://www.fordiac.org.
11. Cengic, G.; Akesson, K., "On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics," *Industrial Informatics, IEEE Transactions on*, vol.6, no.2, pp.145-154, May 2010
12. Colla, M. *et al.* "CEC Designer: Domain Specific Modelling for the Industrial Automation Based on the IEC 61499 Standard", 2008 IEEE Intl Conference on Emerging Technologies and Factory Automation (ETFA'08), Hamburg, September, 2008
13. C. Sünder *et al*, "Execution Models for the IEC 61499 elements Composite Function Block and Subapplication", 5th IEEE Int. Conference on Industrial Informatics (INDIN'07). - Vienna, Austria, 2007. – P.1169-1175.
14. o3neida. IEC 61499 Compliance Profile: Execution Models of IEC 61499 Function Block Applications, [Online] – http://www.oooneida.org/standards_development_Compliance_Profile.html.
15. D. O'Sullivan and D. Heffernan, "VHDL architecture for IEC 61499 function blocks," *Computers & Digital Techniques, IET*, vol. 4, pp. 515-524, 2010
16. Vyatkin, V., "The IEC 61499 Standard and its Semantics" – *IEEE Industrial Electronics Magazine*, 3(4), 2009.
17. ICS Triplex ISaGRAF Workbench for IEC 61499/61131, v.5.1, Online: http://www.icstriplex.com
18. nxtControl. (2010, 10/05). nxtStudio. Available: www.nxtcontrol.com
19. A. Zoitl, Real-Time Execution for IEC 61499: ISA, 2009
20. V. Vyatkin, V. Dubinin, "Refactoring of Execution Control Charts in Basic Function Blocks of the IEC 61499 Standard", *IEEE Transactions on Industrial Informatics*, 2009, doi: 10.1109/TII.2009.2033051
21. Christensen, J.H., „Design patterns for systems engineering with IEC 61499", *Fachtagung "Verteilte Automatisierung - Modelle und Methoden für Entwurf, Verifikation, Engineering und Instrumentierung"*. – Magdeburg, Germany: Otto-von-Guericke-Universität, 2000.
22. G. Booch, I. Jacobson, and J. Rumbaugh. "OMG Unified Modeling Language Specification", Version 1.3 First Edition: March 2000.
23. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: The Semantic Web: Research and Applications. LNCS, vol. 5021, pp. 34--48. Springer, (2008)
24. G. Cengic, K. Åkesson, "On Formal Analysis of IEC 61499 Applications,, Part A: Modeling", *IEEE Transactions on Industrial Informatics*, vol. 6, No. 2, May 2010, pp. 136-144
25. Function Block Development Kit [Online] – http://www.holobloc.com/doc/fbdk/index.htm.

26. Dubinin V., Vyatkin V. "On Definition of a Formal Semantic Model for IEC 61499 Function Blocks", EURASIP Journal on Embedded Systems, Vol. 2008, Article ID 426713. - 10 p.
27. Vyatkin V., Chouinard J., “On comparison of the ISaGRAF implementation of IEC 61499 with FBDK and others implementations”, 6th IEEE Int. Conf on Industrial Informatics (INDIN'08), 2008. – p.1169-1175
28. H. Ehrig *et al*, Fundamentals of Algebraic Graph Transformation, Springer, 2006, XIV, 388 p.
29. AGG Web-site, http: //tfs.cs.tu-berlin.de/agg
30. U. Feige, P. Raghavan. Exact analysis of hot-potato routing, *33rd Annual Symposium on Foundations of Computer Science*, 1992, Pittsburgh, 1992, P. 553 – 562.
31. M. M. Y. Kuo, *et al.*, "Determining the worst-case reaction time of IEC 61499 function blocks," in *8th IEEE International Conference on Industrial Informatics (INDIN'10)*, 2010, pp. 1104-1109.
32. Vyatkin, V., Hanisch, H.M. 'Verification of Distributed Control Systems in Intelligent Manufacturing', *Int. Journal of Intelligent Manufacturing*, Springer, *14*, *(1)*, p. 123-136, 2003
33. K. A. Petri, “Kommunication mit Automaten”, Schriften des Rheinish, Westfalischen Institutes fur Instrumentelle Mathematik und der Universitat Bonn., 1962
34. V. Dubinin, V. Vyatkin. "Graph transformation - based approach to IEC 61499 function block’s formal models synthesis", *Bulletin of Higher Education Institutions, Volga region, Technical Sciences*, Penza State University Publishers, 2008, N 4, P.16-26.
35. V. Dubinin, V. Vyatkin, “Models of Sequential Function Block Execution implemented by dynamically allocated Priorities”, Bulletin of Higher Educational Institutions. Volga Region, ISSN 1728-628X, Penza, 2007, pp.13-22 [in Russian]
36. N. Hagge, B. Wagner, "Modeling and Clarifying the Execution of IEC 61499 Function Blocks Using XNet," in: Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Vienna, Austria, 2007

Victor N. Dubinin received the Diploma in computer science and the Ph.D. degree from the University of Penza, Russia, in 1981 and 1989, respectively. From 1981 to 1989 he was a researcher and from 1989 to 1995 he was a senior lecturer at the same University. Since 1995 he is an associate professor in the Department of Computer Science at the same University. In 2003, 2006 and 2010 he has been awarded DAAD-grants to work as a guest scientist at Martin-Luther-University Halle-Wittenberg, Germany, and in 2011 he held the Visiting Researcher position at the University of Auckland, New Zealand. His research interests include formal methods for specification, verification, synthesis and implementation of distributed and discrete event systems.

Valeriy Vyatkin (SM’04) is Associate Professor and Director of the InfoMechatronics and Industrial Automation lab (MITRA) at the Department of Electrical and Computer Engineering, The University of Auckland, New Zealand. He graduated with the Engineer degree in applied mathematics in 1988 from Taganrog State University of Radio Engineering (TSURE), Taganrog, Russia. Later he received the Ph.D. (1992) and Dr Sci degree (1998) from the same university, and the Dr Eng. Degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999. His previous faculty positions were with Martin Luther University of Halle-Wittenberg in Germany (Senior researcher and lecturer, 1999–2004), and with TSURE (Associate Professor, Professor, 1991–2002).

His research interests are in the area of dependable distributed automation and industrial informatics, including software engineering for industrial automation systems, distributed architectures and multi-agent systems applied in various industry sectors: SmartGrid, material handling, building management systems, reconfigurable manufacturing, etc. Dr Vyatkin is also active in research on dependability provisions for industrial automation systems, such as methods of formal verification and validation, and theoretical algorithms for improving their performance.