# Refactoring of Execution Control Charts in Basic Function Blocks of the IEC 61499 Standard

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

***Abstract*.-This paper deals with refactoring of execution control charts of IEC 61499 basic function blocks as a means to improve the engineering support potential of the standard in development of industrial control applications. The main purpose of the refactoring is removal of arcs without event inputs. Extended refactoring, proposed in the paper, also helps to get rid of potential deadlock states. The ECC refactoring is implemented as a set of graph transformation rules. A prototype has been implemented using the AGG software tool. The refactoring can help in implementing equivalent transformation of control programs without introducing errors.**

***Index terms* – IEC 61499, refactoring, software engineering, graph grammars**

## I. Introduction

The international standard *IEC* 61499 [1] defines a component-based architecture for the new generation of distributed component control systems. This standard is considered by many researchers and practitioners as the key enabler for improving flexibility and reconfigurability of automated manufacturing systems. The standard introduces modern component and visual programming ideas to the industrial automation world. In particular, the standard promotes the idea of using communicating state machines for programming automation systems. The main construct of the standard's architecture is *function block* (FB). The Execution Control Chart (*ECC*) is a state machine determining sequence of operations in a basic FB.

After the final approval by the International Electrotechnical Commission in 2005, the IEC 61499 standard is vigorously finding its way to the industrial automation practice. There are several commercial and academically developed tools, along with a reasonable number of pilot installations [11]. Size and complexity of control programs implemented in function blocks has grown significantly, and the problem of design support by efficient computer-aided engineering tools is of paramount importance.

System engineering with function blocks has much in common with object- and component-oriented design in the general software engineering and many ideas and concepts aiming at the code quality improvement can be borrowed from there. One such technique that has become important in software engineering in the recent years is *refactoring* [2]. Refactoring changes program structure without changing its semantics. Refactoring is a technique supporting evolution of software systems, which can be applied to different abstraction levels of software models – from low-level code up to high level models.

*Model Driven Engineering – (MDE)* is one of the state-of-the-art software engineering technologies, and it operates with models and their transformations [3]. The Object Management Group (OMG) [4] has proposed the Model Driven Architecture (MDA) for integration of various *MDE* tools. For definition of models and metamodels the *OMG* consortium has developed popular standards *MOF* and *UML*. In [5] an approach, called Model-Integrated Computing (MIC) for expanding *MDA* into the field of domain-specific modelling languages, is proposed. In particular, the MIC-approach was applied in [6] in the area of mechatronic systems.

Graph transformations [7] are a promising technique of implementing model transformations, as confirmed by its application in *MDE*, e.g. [8]. They also can be used for refactoring of program structures represented by graphs. This becomes especially important with the progress of visual programming methods. A good introduction to refactoring using graph transformations can be found in [9].

According to us, this approach is also appropriate for use in engineering of function block systems [10]. Main artefacts of the standard's architecture, such as composite FBs, applications and subapplications, can be represented in an abstract graph form. This also applies to basic FBs whose Execution Control Chart can be naturally represented as a graph.

One problem constantly present in discrete control design is deadlock avoidance. A poorly designed controller can come to a *deadlock* state that it cannot leave at any further input. The state-machine based programming approach of IEC 61499 provides an opportunity to solve this problem, at least partially, by applying model-transformation techniques. It does not make obsolete other approaches, such as formal verification based on reachability analysis (addressed by many researchers, from [12] to the recent [13]), but the latter are capable of only detection but not correction of deadlocks, and are a lot more complicated for use by control systems developers.

The importance of deadlock (or livelock, i.e. infinite loop) avoidance has been recognized by the practitioners. Thus the latest version of the FBDK software tool [14] recognizes and

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)

V. Dubinin is with the Department of Computer Science, University of Penza, Penza, Russia (e-mail: victor_n_dubinin@yahoo.com )

This work was supported, in part, by the research grant FRDF 3622763 / 9573 of the University of Auckland.

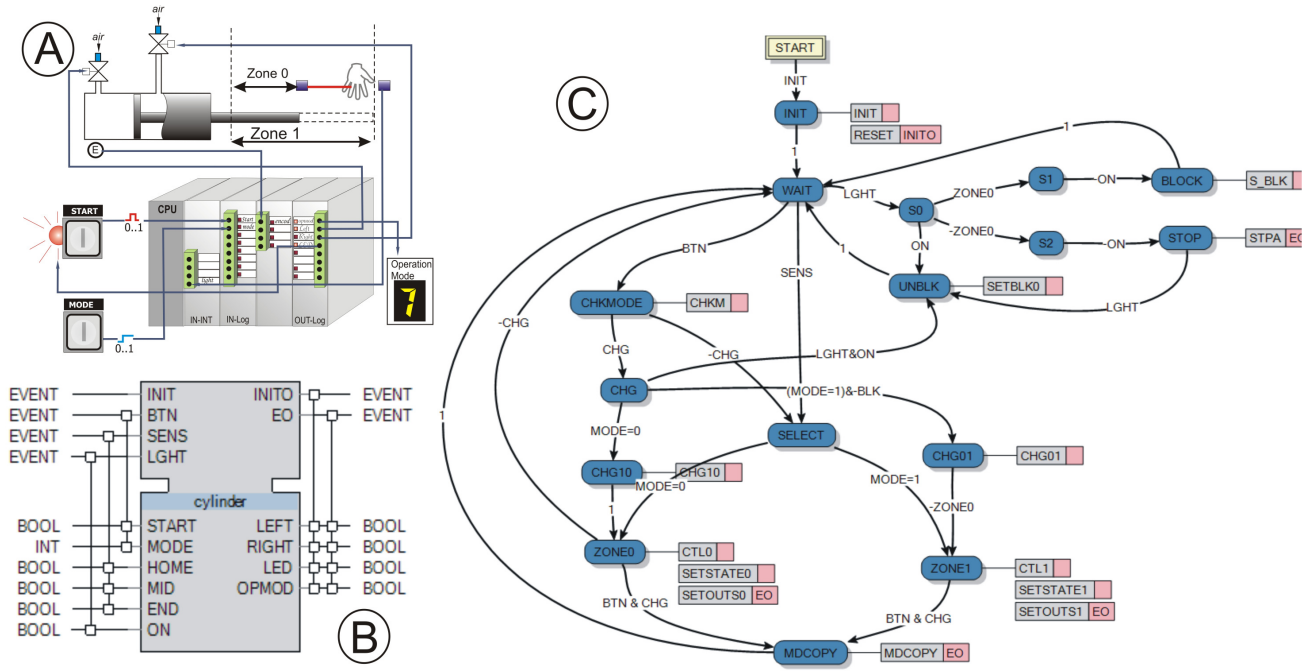



Figure 1. Pneumatic cylinder-based control system (A); interface of the controller function block (B); ECC of the controller function block (C).

prohibits creation of ECCs with simple loops of eventless transitions. This simple measure, however, cannot guarantee complete avoidance of incorrect situations in ECC.

In this paper we develop graph transformations-based refactoring methods aiming to get rid of ECC arcs having no event conditions. This leads to a technique of ECC improvement allowing the removal of (conditionally) dead states. We present and classify graph transformation rules for the ECC refactoring. The prototype refactoring system is implemented in the graph transformation tool AGG [15, 16].

The paper is structured as follows. Section II presents a motivating example from industrial automation domain. In Section III, a formal model of ECC syntax is introduced. Section IV discusses ECC execution rules to the extent relevant to this paper. The concept of ECC refactoring is defined in Section V. Section VI presents the idea of refactoring implementation by means of graph transformations, and Section VII discusses transformation rules in detail. Section VIII shows how the system of transformation rules was implemented using the AGG software tool. Section IX presents evaluations of the developed refactoring technique. The paper is concluded with a short summary of the presented work, outlook and references.

## II. Illustrative Example

For illustration of the deadlock problem and of the proposed solution we will use a simple example - pneumatic cylinder with some control buttons and light curtain safety device, as presented in Figure 1,A. Controller of this system is implemented as an IEC 61499 function block “cylinder”. Its interface is shown in Figure 1, B, and the control logic, implemented as a state machine (ECC), in Figure 1, C.

The operation is as follows. The cylinder shuttles back and forth either from the left to the middle position or from the left to the end position depending on the selected mode of operation. The mode is selected by pressing the button “MODE” which has two fixed positions, one corresponding to the value 0 and the other to the value 1. When any object crosses the safety light curtain the operation has to stop until the object leaves the safety zone.

The light curtain signal is connected to a specific input port of the control device that generates interrupt at every change of the value. In terms of function blocks, the interrupt is translated to the event input LGHT of the “cylinder” FB.

This FB has six logic inputs, corresponding to START and MODE buttons, 3 discrete position values (HOME, MID, END) and the logic status of the light curtain (ON). Also there are 4 event inputs. The INIT is used for the FB initialisation. The BTN event input indicates a change in a button state (pressed/released), the SENS event input is raised when the cylinder arrives to one of the three discrete positions. The LGHT event input indicates a change in the light curtain status.

Output signals of the “cylinder” FB are: actuators LEFT and RIGHT, and two indicators: LED for lighting the button START in those times of operation when it needs to be “sensitive” to a hit, and OPMODE, used to display current operation mode (i.e. zone 0 or 1).

The controller state machine in Figure 1,C combines the sequential logic (implementing the back and forth movement) and reaction to interrupts. Substantial parts of control logic are encapsulated into the algorithms executed in ECC states. For example CTL0 and CTL1 algorithms (states ZONE0 and

ZONE1) are written in the Ladder Diagrams language. Their code is not presented here for the sake of brevity, but their main function is to recalculate actuators' logic outputs.

One should note that there is no established designed methodology for design of such event driven state-machine based controllers, so this particular design cannot be regarded as anyhow typical. It represents a design effort of an average engineer.

It comes at no surprise that this state machine has some deadlock states. For example, after an interruption from the light curtain occurs (as a result of an "invasion" while the shaft is still in Zone 0), the ECC goes through the state trace WAIT→S0→S1→BLOCK→WAIT, setting the internal variable BLK to the value TRUE (this value is supposed to be checked if the operator changes the MODE to 1). However, after the invading object has been removed, the ECC will go through WAIT→S0→S1 and stop in S1 forever, even though ON=1. This happens because the arc S0→S1 has higher priority than S0→UNBLK.

It is quite obvious that real automated machines may include dozens of the processes similar to the cylinder's operation, so their controller state machines will be a lot more complex and it will be even more difficult to find and fix them manually.

## III. Model of Execution Control Chart

To explain our refactoring approach we need to introduce some formal notation of *ECC* that is simplified from the more comprehensive model of [17].

An *ECC* can be defined as a tuple:

$ECC = (S, R, E, C, A, D, f_E, f_C, f_A, f_P)$, where

$S = \{s_1, s_2, \ldots, s_n\}$ is a set of vertices representing *EC*-states;

$R \subseteq S \times S$ is a set of arcs representing *EC*-transitions;

$E=\{e_1,e_2,\ldots,e_m\}$ is a set of event inputs;

$C=\{c_1,c_2,\ldots,c_k\}$ is a set of guard conditions defined over input, internal and output variables of a basic FB;

$A=\{a_1,a_2,\ldots,a_p\}$ is a set of *EC*-actions' sequences.

$D \subseteq A \times C$ is a relation, defining *dependency* of transition conditions on the results of *EC*-actions, $(a_i,c_j)\in D$, if the execution of $a_i$ can change the evaluation of $c_j$. It should be noted that, as the practice shows, the dependence of guard conditions on *EC*-actions happens quite seldom.

The set of arcs $R$ is divided into three classes: $R_E$ - event, $R_C$ – conditional, $R_T$ - unconditional arcs, such that:

$R= R_E \cup R_C \cup R_T$; $R_E \cap R_C \cap R_T=\varnothing$.

The syntax of *EC*-transition conditions is defined as: *Event input | Guard condition w/out event inputs | Event input & Guard condition.*

In our model, an *EC* transition is represented by an arc of one of the following types: an event arc *(E*-arc*)* represents *EC*-transition with event input in its condition; a conditional arc *(C*-arc*)* represents an *EC*-transition without event input whose guard condition is not constantly *TRUE*; and unconditional arc *(T*-arc*)* represents an EC transition without event input and with the constantly *TRUE* guard condition. In the graphical notation, *E*- and *T*-arcs will be depicted by a solid line and *C*-arcs by a dashed line. When necessary, in drawings we shall put symbol *"t"* above T-arcs and symbol *"e"* above *E*-arcs.

$f_E$: $R_E \rightarrow E$ – the function assigning event inputs to *E*-arcs;

$f_C$: $R_E \cup R_C \rightarrow C$ – the function assigning guard conditions to *E*- and *C*-arcs;

$f_A$: $S \rightarrow A$ - the function assigning sequences of *EC*-actions to the states.

$f_P$: $R \rightarrow \{1,2,\ldots,\}$- the function assigning normalized priorities of arcs, defined for the whole ECC as $f_P = \bigcup_{s\in S} f_P^s$, where $f_P^s: R^s \rightarrow \{1,2,\ldots,|R^s|\}$ is the function of prioritization for the vertex $s$, and $R^s$ is the set of all arcs which are starting in the vertex $s$. The priority of an arc $r_1$ is higher than of $r_2$ if $R(r_1)<R(r_2)$.

It must be noted that in *ECC* of *IEC* 61499 priorities of *EC*-transitions are not defined explicitly, instead, the priority is based on the location of the transition in the textual representation of the function block (in the XML format).

## IV. Models of ECC execution

The *IEC* 61499 defines some rules of *ECC* interpretation.

The *ECC* interpreter is activated by an input event and continues evaluation of *ECC* until no *EC*-transition can clear (i.e. evaluate to *TRUE*). This process may include several *EC*-transitions and is called *a single run* of FB, and the sequence of actions executed during a single run is called a *trace* of the *ECC*. However, as it was noted in [18, 19], the definition of *ECC* interpretation in the standard is incomplete and, thus, ambiguous. For example, it admits two different approaches to evaluation of *EC*-transitions without events.

According to the first approach, an *EC*-transition without events can be cleared only if it is not first in the run, but follows some other *EC* transition with an event name in its condition. The second approach does not link *EC*-transition to any concrete event. In this case enableness of the *EC*-transition is determined only by the value of its guard condition. We shall name an eventless guard condition *passive* in the first case and *active* in the second case. Both approaches were studied in the literature. The first approach is presented in [19], and the second is presented in the work introducing the sequential model of FB execution [18]. In the following we shall consider only the first model of *ECC* realization that represents a more compelling case for the proposed refactoring of *ECC*.

Further we define some essential concepts in a semi-formal way as follows:

*Definition 1.* An *ECC* state is called *potentially deadlock* (PD) state if all its outgoing arcs are conditional.

*Definition 2.* Two *ECCs* are called *functionally equivalent* (within the limits of a particular model of ECC execution), if in any initial state and at any sequence of input events and corresponding values of input variables, both *ECCs* produce same traces, i.e. execute same sequences of *EC*-actions.

## V. Refactoring and Improvement of ECC

The goals of ECC refactoring are to get rid of *C*-arcs and *PD*-states completely, if possible, or, at least, to minimize the number of *C*-arcs and to delete *PD*-states emerging as a result of this minimization. According to these goals we will introduce two types of refactoring (type 1 and 2 respectively). Refactoring-1 can help the developer to have a different point of view on the developed *ECC* that in some cases can help to rethink and redesign it. Refactoring-2 goes further and improves the *ECC* by removing potential deadlocks. It is applied on top of the refactoring-1.

Let us name as *CT*-network of an *ECC* a subgraph containing arcs only from $R_C \cup R_T$, but not from $R_E$. In general, such a graph may not necessarily be connected. Accordingly, as *T*-network of *ECC* we shall name a subgraph containing only the arcs from $R_T$.

It is assumed that the initial *CT*-network is acyclic. Presence of cycles in the *CT*-network tells about incorrectness of the *ECC*. Although in the general-purpose programming cyclic structures are of wide use, in ECCs of function blocks it is recommended to implement iterative procedures in algorithms rather than in ECC.

Let us introduce $ES = \{(s, s') \in R_E | \exists (s', s'') \in R_C \cup R_T\}$ – the set of *E*-arcs having a *C*- or *T*-arc as a 'successor'. These arcs, referred to as *sources*, will be the main starting points of the refactoring actions introduced further. The general idea of removing *C*-arcs from *ECC* is as follows. Let $(s_0, s_1) \in ES$ be an *E*-arc followed by the path $s_1, s_2, \ldots, s_k$ in the *CT*-network. For each *EC*-state $s_i$ ($i$=1,..., $k$) there is a sequence of associated *EC*-actions $a_i$. An example is given in Figure 2, where the path w.l.o.g. consists of *C*-arcs only.

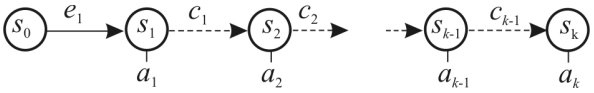



Figure 2. A path consisting of a source arc followed by C-arcs.

In case when transitions' conditions are independent on their preceding EC-actions in a single run, this path can be substituted by one *E*-arc $(s_0, s_k)$ with guard condition being a conjunction of the guard conditions of arcs $c_i$, ($i$=1,..., $k$) and of the condition $q$, called *condition of the state preservation* (Figure 3).

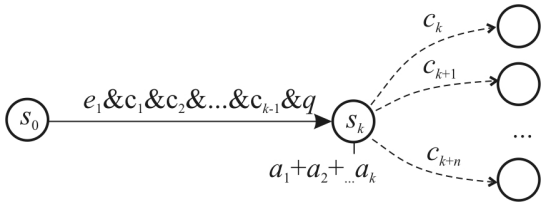



Figure 3. An *E*-arc representing the path from Figure 2.

The sequence of *EC*-actions executed in the target state $s_k$, is derived as a concatenation of all EC-actions' sequences across the vertices forming the path. The condition $q$ of a state $s_k$ preservation is defined as a conjunction of the guard conditions' negations across the outgoing *C*-arcs: $\&_{(s_k, s_j) \in R_C} \overline{f_C(s_k, s_j)}$. For example, for the state $s_k$ Figure 3 the condition of the state preservation is equal to $\overline{c_k} \,\&\, \overline{c_{k+1}} \,\&\, \ldots \,\&\, \overline{c_{k+n}}$.

If there are more incoming arcs to the final state $s_k$ of the path $s_0, s_1, \ldots, s_k$, then the path needs to be substituted by two arcs $(s_0, s_{k-1})$ and $(s_{k-1}, s_k)$, first of which is identical to the *E*-arc from Figure 3, the second being a *T*-arc. This is illustrated in Figure 4. The second arc is needed since it would not be correct to assign the whole sequence of actions $a_1+a_2+\ldots+a_k$ to $s_k$, due to other paths possibly ending there. Instead, we assign almost the whole sequence of actions (but the last $a_k$) to the vertex $s_{k-1}$, further referred to as a *proxy* of the vertex $s_k$.

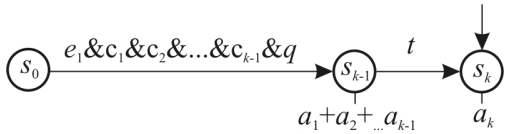



Figure 4. *E*-and the *T*-arcs representing the path from Figure 3.

For a given arc $r=(s_i, s_j) \in ES$ we introduce *binding* operation with an arbitrary vertex $s_k$ from the *CT*-network. The operation consist in finding all paths from $s_j$ to $s_k$ in the *CT*-network, and in substituting them by *E*-arcs (or by (*E, T*) pairs of arcs) as described above and illustrated in Figure 4. In general, the outcome of such an operation is a so called *hammock graph* as the one in Figure 5. All *E*-arcs going out of $s_i$ have the same event input name in their condition $f_E(s_i, s_j)$ ($f_E(s_i, s_j)=e_m$ in the Figure). It must be noted, however, that the binding operation is not always applicable.

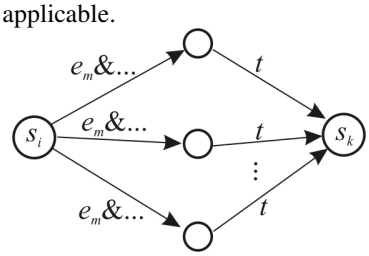



Figure 5. Outcome of binding a source arc and a vertex of the CT-network.

The binding of an arc $r_i \in ES$ with all vertices of the *CT*-network will be referred to as *binding* of this arc by the *CT*-network. For complete removal of *C*-arcs from *ECC*, it is necessary to bind all arcs from the *ES* set with the corresponding *CT*-network and then to delete all *C*-arcs.

It is possible to prove that any acyclic *CT*-network without dependencies between EC-actions and guard conditions (i.e. at $D=\varnothing$) can be made *C*-arcs free as a result of such transformations. The resulting *T*-network in combination with *E*-arcs can be called reachability graph of the *EC*-actions' sequences in the original *CT*-network. The ECC, obtained as a result of such transformations, is, obviously, functionally equivalent to the original ECC.

When doing refactoring, it is important not only to obtain new ECC structure, guard conditions and EC-actions' sequences, but also to determine priorities of arcs in the

resulting ECC. We present the method for determining arcs' priorities on example of an *ECC* having a binary tree form (Figure 6). We will refer to *C*-arcs by their guard conditions.

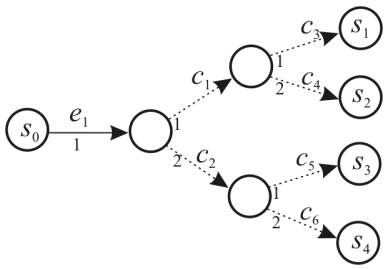



Figure 6. An example of *ECC* in the binary tree form.

The initial (normalized) priorities annotate the corresponding arcs. It is obvious, that the path $c_1$, $c_3$ has the highest priority, as it will be chosen by the ECC interpreter if all the conditions $c_1$, …,$c_6$ are TRUE. The path $c_2$, $c_6$ – has the lowest priority. It will be chosen only if conditions $c_2$ and $c_6$ are TRUE, and all others are FALSE. From this example one can derive an idea of a simple rule for priorities assignment such that the influence of an arc on the overall path priority is the higher the nearer the arc is located to the path's beginning. Based on this, we propose to use composite priorities (formed as a tuple) with the lexicographic order defined on them. The composite priority is formed as a concatenation of arcs' priorities in a path from its initial vertex up to the end. The resulting assignment of composite priorities is presented in Figure 7, the priorities are written under the corresponding arcs.

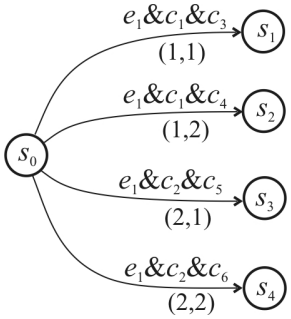



Figure 7. Converted *ECC* (from the ECC in Figure 6) with composite priorities of arcs.

## VI. Implementation of Refactoring by Graph Transformations

In the following, we present a mechanism for implementation of the proposed refactoring methods based on equivalent transformations of ECC using a typed attributed graph rewriting system. One equivalent transformation can consist, in general, in application of several *transformation rules*.

There are several approaches to graph rewriting, one of which is the algebraic approach. The algebraic approach is divided into three sub-approaches: the double-pushout approach (DPO), the single-pushout approach (SPO), and the pullback approach [7,20]. We briefly consider the first ones mainly because of using AGG tool [15] as an implementation platform for the refactoring.

Let us briefly introduce some terminology from the theory of graphs' transformations according to SPO [7]. Let $L$ and $R$ be labelled graphs. A graph production rule is a morphism $p$:$L\rightarrow R$. A direct graph transformation $G\Rightarrow_t H$ (of graph $G$ to graph $H$) is a pair $t=(p,m)$, consisting of a graph production rule $p$:$L\rightarrow R$ and an injective graph morphism (called *match*) $m$:$L\rightarrow G$. Given a direct graph transformation (i.e. the pair of two morphisms $p$ and $m$), it is possible to derive the morphisms $m'$:$R\rightarrow H$ and $p'$:$G\rightarrow H$, as illustrated in the pushout diagram in Figure 8 . In practical terms, the $p'$ morphism is the one, needed to generate graph $H$ being the result of the transformation.

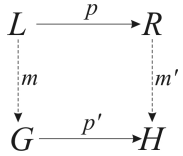



Figure 8. Schematic representation of a direct graph transformation.

A sequence $G_0\Rightarrow G_1\Rightarrow \ldots \Rightarrow G_n$ of direct graph transformations is termed as *a graph transformation* and is designated $G_0\Rightarrow^* G_n$.

The condition of non-applicability (*NA*-condition) of a rule $p$ is a graph morphism $nac$:$L\rightarrow L'$. A direct graph transformation $G\Rightarrow_{(p,m)} H$ satisfies a *NA*-condition if there is no graph morphism $m'$:$L'\rightarrow G$ exists such that $m'\circ nac = m$. In simple words, the *NA*-condition is a graph which determines a forbidden graph structure. One transformation rule can have several associated *NA*-conditions. In this case, a rule is applicable if all the *NA*-conditions are satisfied.

An *attributed graph* is a graph, whose vertices and arcs are marked by abstract data types. In case of attributed graphs, for applicability of a transformation rule the fulfilment of conditions on attributes of vertices and arcs is also required (if there any). When a rule is applied, values of attributes in a certain part of the resulting graph can be recalculated. More detailed information on transformation of typed attributed graphs can be found in [20].

## VII. Transformation Rules

Basic transformation rules of a refactoring system transform a pair of adjacent arcs $(s_i,s_j)$ and $(s_j,s_k)$ to a new direct arc, leading to the state $s_k$ or to its "proxy". In our refactoring *ECC* system, most of the rules aim at construction of a set of *EC*-action sequences reachable by paths of length 2 at an occurrence of some event. The arcs, entering and going out of the vertices $s_i$, $s_j$ and $s_k$ (except the two arcs $(s_i,s_j)$ and $(s_j,s_k)$), represent the context of the rule's application.

The proposed *ECC* refactoring system consists at the moment of 35 rules that can be divided into the following classes:

1) rules of preliminary graph correction;

2) rules of graph increment; and

3) rules of graph clearing.

The algorithm of the rule based *ECC* transformation is quite straightforward:

1. First, rules of the first class are applied as long as it is possible. For that a match is being sought between the source ECC and the left part of each rule. If the match is found, the corresponding subgraph of the ECC will be transformed into the subgraph in the right-hand side of the rule.

2. Then, rules of the second class are applied in the same manner.
3. Finally rules of the third class are applied.

Rules of the first class are needed to remove some obsolete arcs from the source *ECC*. Examples of this type of rules are presented in Figure 9-11. The rule of parallel *C*-arcs merge removes the presence of several unidirectional *C*-arcs between the pairs of *EC*-states (Figure 9).

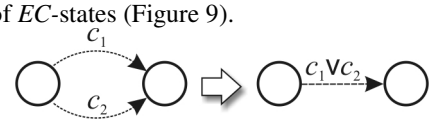



Figure 9. The rule of merging parallel C-arcs.

The rule of dead *E*-arcs removal eliminates the arcs going out of an *EC*-state being the origin of at least one *T*-arc (Figure 10). These *E*-arcs will never be passed, otherwise the *ECC* will immediately jump from the source *EC*-state to the target *EC*-state of the *T*-arc.

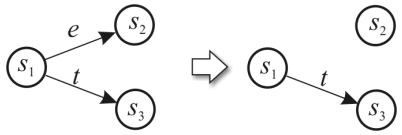



Figure 10. The rule of deleting dead *E*-arcs.

The rule in Figure 11 deletes a *T*-arc which is dead because it has a lower priority than another *T*-arc going out of the same *EC*-state. Priorities of arcs are designated as *pr*. The condition of the rule's application is written above the arrows connecting the left and the right parts of the rule, e.g.: $x<y$.

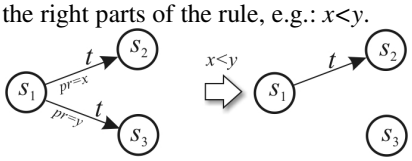



Figure 11. The rule of removal of a dead T-arc with a lower priority.

Rules of the second group (graph increment) perform the main part of *ECC* transformation. They include not only removal of arcs, but also adding of new nodes and arcs and modification of nodes' and arcs' attributes.

In Figure 12-16 some rules of this class are illustrated. One should note that rules of this class are subdivided into subgroups based on similar functionality. Inside each subgroup the rules differ only by their context. We will refer to the whole subgroups as R1, R2,…, but will illustrate only one rule from each subgroup.

The graphical notation is as follows. Context states are represented by smaller circles. The type of context arcs is not specified. The cross on a context arc in the left part of a rule means that the arc is prohibited. This way *NA*-conditions are concisely represented. The dash on an arc in the right side of a rule means that the arc has been used and will be removed. Two conditions are calculated for an *E*-arc: 1) condition of reaching a target state (including also the name of the corresponding event input), and 2) a condition of signal propagation beyond the target state. The first condition is written above, and the second one under the arc. The complete current condition for an *E*-arc is defined as a conjunction of the first condition and of the negation of the second condition. The second condition is omitted if its value is *false*.

The goal of the rule R1 (Figure 12) is to remove condition arc $(s_2,s_3)$ that follows an event arc. This is achieved by adding direct arc $(s_1,s_3)$, modifying condition under $(s_1,s_2)$, and transferring the actions of $s_2$ to $s_3$. This rule can be applied to such ECC parts where $s_2$ and $s_3$ vertices have no incoming arcs (as indicated by the context arcs with cross in the left part of the rule).

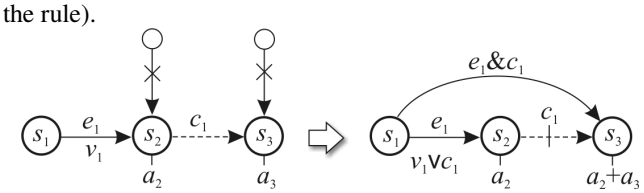



Figure 12. Rule of an event propagation on a linear section (R1).

Rule R2 (Figure 13) has a similar goal, but can be applied to *ECC*'s parts in which $s_3$ has incoming arc. To avoid conflicts, that can potentially arise when actions are "transferred" to this vertex from different paths, an intermediate state $s_4$ is introduced, where the actions of $s_2$ are assigned to.

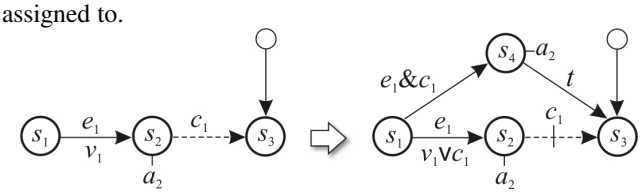



Figure 13. Rule of entrance into a connector (R2).

The goal of the rule R3 is to make a "clear path" between two vertices ($s_1$ and $s_3$), i.e. get rid of the arc coming into the intermediate node between these vertices.

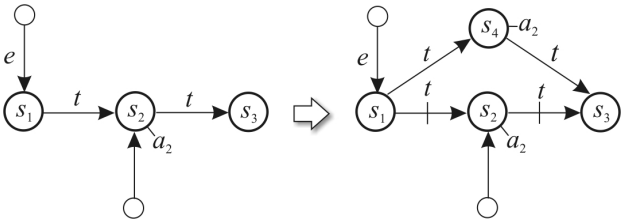



Figure 14. Rule of a connector bypass (R3).

As a rule, a newly created (daughter) arc has two parents – the pairs of adjacent arcs. Thus the priority of the daughter arc is defined as a concatenation of the parent arcs' priorities in the order following the order of parents.

The application condition for the rules *R1*, *R2*, and *R4* shown in Figures 12, 13, and 15, respectively, is absence of dependencies between actions and *EC*-transition conditions, i.e. $(a_2,c_1)\notin D$.

In the rule removing inverse *C*-arcs (Figure 15), the right side, unlike most of other rules, contains two daughter arcs ($s_1$, $s_3$) and ($s_1$, $s_2$), having equal priority, same as the priority $x$ of their parent arc. The priorities do not matter in this case as these arcs are mutually exclusive w.r.t. the condition $c_1$.

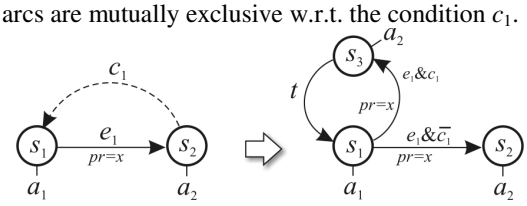



Figure 15. Rule of inverse *C*-arc removal (R4).

Unlike the rules of the first and second groups, the rules of graph clearing are specific to each of the refactoring types. For example, the rules of an E-arc removal and of an isolated vertex removal are shown in Figure 16 a) and b) respectively.

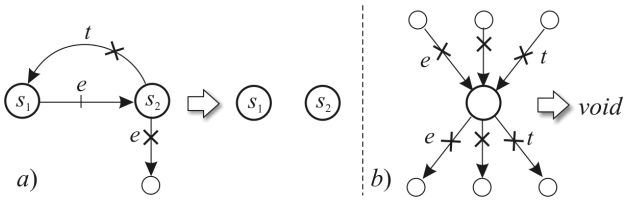



Figure 16. Rules of deleting a) an E-arc (R5); b) an isolated vertex (R6).

According to the first rule, an *E*-arc ($s_1$,$s_2$) is deleted IF it has been used for generation of other arcs AND is not incident to other *E*-arc AND there is no reversed *T*-arc ($s_2$,$s_1$).

To illustrate the process of refactoring we consider transformation of a simple *ECC* in Figure 17,(a). In a simplified form, the refactoring-1 process is defined by the sequence of rules *R1, R1*. The simplification (introduced to facilitate understanding) consists in eliminating *C*-arcs directly in the rule *R1* (instead of applying a separate rule). The first application of *R1* results in an intermediate *ECC*, functionally equivalent to the original ECC (Figure 17,(b)). The next *R1* application produces the result of the refactoring-1 (Figure 17,(c)) which is also semantically equivalent to the initial *ECC*. In the resulting *ECC* the PD-states are presented in the form of deadlock vertices $s_1$ and $s_2$.

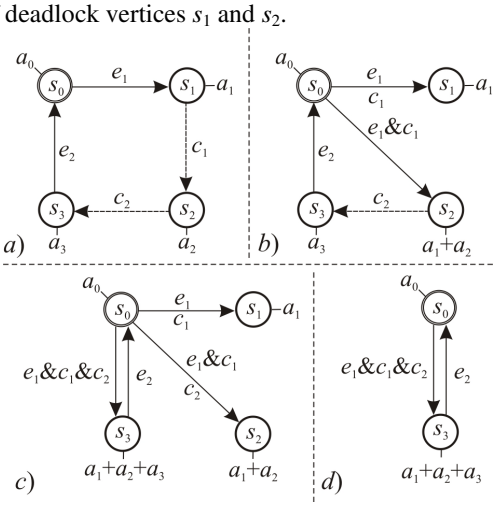



Figure 17.a)A sample ECC; b) Intermediate result of *ECC* transformation; c)The result of refactoring-1; d) The result of refactoring-2.

Refactoring-2 is done by additional application of rules *R5* and *R6*. The resulting *ECC* for this case is presented in Figure 17,(d). This *ECC* is not semantically equivalent to the initial *ECC* as it does not have *PD*-states $s_1$ and $s_2$. As a result, starting from the initial state $s_0$ at event $e_1$ and when $c_1$=*TRUE* and $c_2$=FALSE (i.e., $e_1 \& c_1 \& \overline{c_2}$), no EC-actions will be executed in the resulting *ECC*, while the original *ECC* would execute the sequence of actions $a_1$ and $a_2$ and then would freeze. In spite of the fact that initial and resulting *ECCs* are not equivalent, the resulting *ECC* in Figure 17,(d) most likely matches the intentions of the developer. In this particular case, the original specification (or intention of the designer) may have been to wait until event e1 occurs, then execute a1, then wait until c1 is true and execute a2, and so on. Results of the refactoring show that the designed ECC does not achieve this goal. The spotted deadlock states may have appeared because of imprecise understanding by the designer of the ECC semantics that is different from the general finite automata semantics. Priorities in the example in Figure 17 do not matter due to the mutually exclusive guard conditions.

Application of even some of the refactoring rules can have practical importance. Thus, if the merger of consecutive *C*-arcs is applied to the ECC of the "cylinder"[2], the result will be free from the deadlock states $s_1$, $s_2$ as shown in Figure 18.

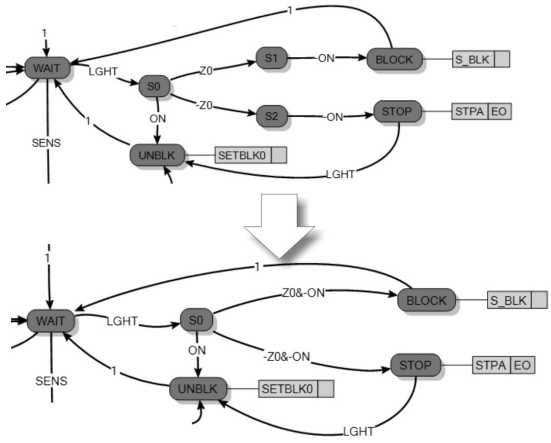



Figure 18. Application of the C-arcs merger rule to the ECC of "cylinder" FB helps to get rid of deadlock states S1 and S2.

[2] This group of rules is not illustrated in the paper due to the lack of space.

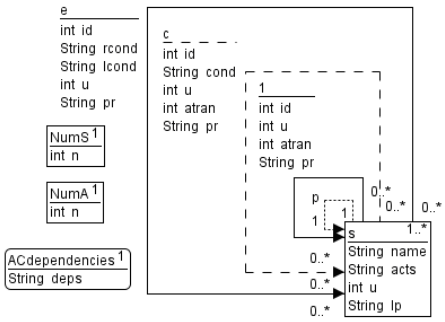



Figure 19. ECC metamodel.

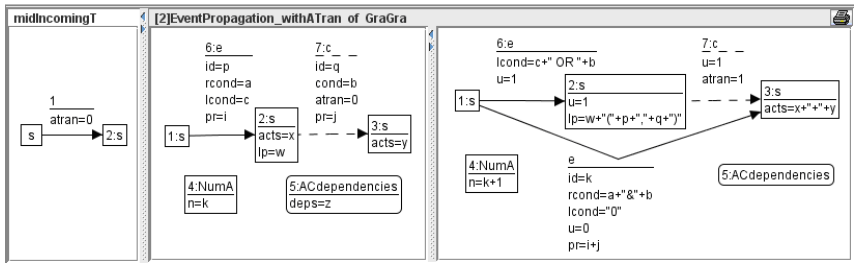



Figure 20. The rule of event propagation (R1, Figure 13), presented in AGG.

## VIII. IMPLEMENTATION OF REFACTORING IN THE GRAPH TRANSFORMATION ENVIRONMENT AGG

A software prototype of the *ECC* refactoring system has been developed using the graph transformation environment *AGG*. *AGG* is a rule-based visual programming environment, based on the use of the algebraic single push-out approach to graphs transformation [15, 16]. The rules of graph transformation can contain *NA*- and application conditions. *AGG* supports specification of typed graphs with cardinality and attributes. Types of attributes are borrowed from the *Java* programming language.

As shown in Figure 19, the metamodel of *ECC*, used in the refactoring system prototype, consist of a single node of *S* type (*EC*-state) and four loops, representing *E*-, *C*-, *T*-arcs, and *p*-arcs correspondingly. The arcs of type *p* are auxiliary and used usually for linking parent node and newly created daughter node. The metamodel is represented in the form of type graph and used in AGG as a tool for control of correctness of graph transformation. An EC-state has the following attributes: *name* - a name *of EC-state*, *acts* - a list of associated *EC*-actions, *lp* - the list of arc pairs used in the generation of arcs through the given vertex. Some attributes are common for all arcs, but there are some specific attributes. All arcs have the following common attributes: *id* – a unique identifier, *u* - an indication of usage (*u=1* - the arc has been used, *u=0* - not used), *pr* - priority. *C*- and *T*-arcs have the attribute *atran* - an indication of transmission of actions on the arc (*atran=1* – actions were passed, *atran=0* – not passed). The attribute *cond* of a *C*-arc defines its guard condition. For an *E-arc*, two conditions are attached: a condition of reaching the arc's target state (attribute *rcond*), and a condition of leaving the target state (attribute *lcond*). As a rule, the guard condition of an *E-arc* is formed as a conjunction of *rcond* and negation of *lcond*. To simplify implementation of the composite priorities mechanism, it is assumed that they are of a character type. For deriving the composite priority, the concatenation of the strings is used. The relation of lexicographic order is implemented by the operation of strings comparison in the *Java* language.

Three types of auxiliary nodes are used in the system. The node of type *NumS* is used for indexing of newly created vertices, the node of type *NumA* - for indexing newly created arcs, and the node of type *ACdependencies* contains pre-defined "database" of existing *Action-Condition* dependencies.

The refactoring system comprises some 35 rule divided into 5 layers. In Figure 20 the *AGG* implementation of the rule from Figure 13 is shown. The left part of the rule is presented in the middle pane, and the right part in the rightmost pane. One of the *NA*-conditions is presented in the left part of the window, there are 4 more *NA*-conditions which are not shown. The shown rule has the following application conditions expressed in *Java*: 1) *w.indexOf* (*"("+p+","+q+")"*) < 0; 2) *!My.isdep(x,b,z)*. The first condition enables the rule if the arcs 6 and 7 have not been used earlier for generation of another arc. The second condition determines absence of dependences between *EC*-actions from *x* and guard conditions from *b* in the existing *AC*-dependencies *z*. Here *My* is an user-defined Java class and *isdep* is a method of this class implementing the check.

Graph transformations artefacts in the *AGG* system are saved in a special *XML*-based format (*GGX*-format). To use AGG in the FB tool chain we have developed two convertors from/to the standardized XML representation of function blocks into AGG format GGX. With these convertors one can import a function block into AGG, apply refactoring and then export the result back to XML. The converters can help to integrate refactoring into the corresponding IEC61499 software engineering tools.

Another implementation idea for the developed method may rely on re-implementation of the transformation rules in advanced system engineering tools for IEC 61499. For example, there are several open-source projects aiming at the development of such tools, e.g. 4DIAC-IDE [23] and FBench [24], which can facilitate integration of refactoring into the system engineering practice. The role of AGG in this scenario would be to develop and verify transformation rules. Once the rules are re-implemented in an engineering support environment, the environment can provide a user – friendly way of rules' application, say with an option of seeing the result of a transformation and accepting/undoing it. This way an advice can be given to the engineer on how to improve the code and even make it deadlock free.

## IX. EVALUATION

The visual (graph-based) representation of rules and transformed systems is the most intuitive form that helps to avoid errors. However, the main advantage of using graph transformations for refactoring is that properties of graph transformation systems have been well studied theoretically. As our system makes a particular case, many of the existing theoretical results are directly applicable.

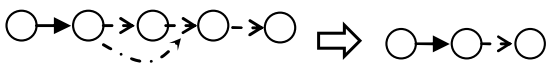
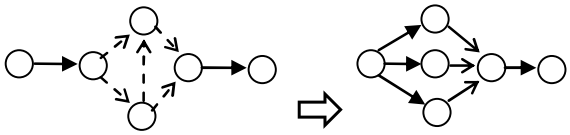
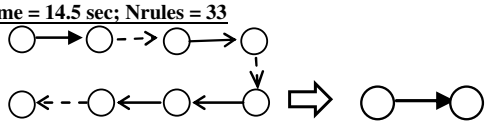
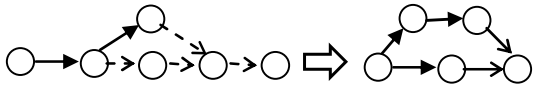
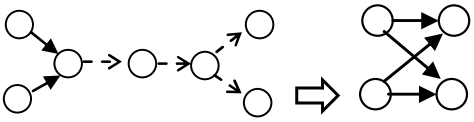
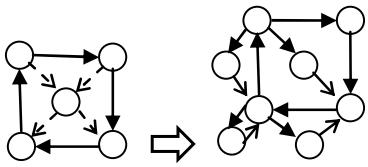
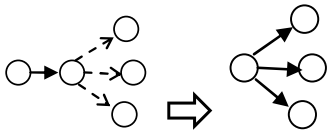
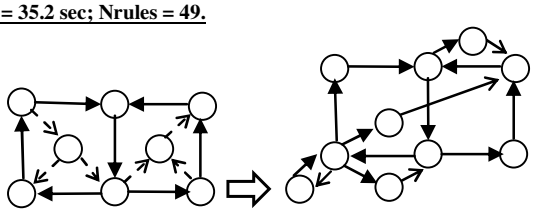
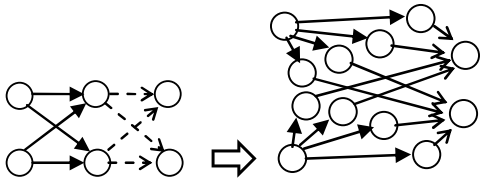
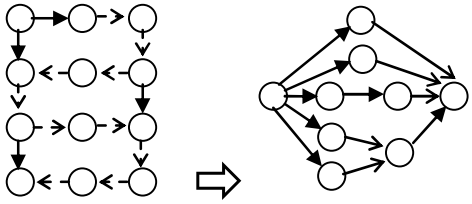
When analyzing the developed refactoring system, such properties as complexity, completeness, correctness, and confluence are of interest.

The computational complexity of the graph rewriting application, in general, is high, since it is based on matching of subgraphs which is a known *NP*-complete problem. However, in our case the sub-graphs in the left part of the rules are quite small. Moreover, node and edge labels, as well as directed edges drastically reduce the search space for isomorphic subgraphs.

The system of refactoring rules considered in this paper has been tested using more than 30 *ECC*s representing various typical structures. Results of some tests are presented in Table 1. The tests were created empirically, following basic graph topologies, such as linear sequence, branching, connector, each-to-each, etc. We tried to exclude topologies similar one to another, e.g. triangle is similar to rectangle, so it was excluded from the test set. Some of the tests represent regular structures in the graph algebra, so different basic features can be combined into more complex ones. However, we also tested irregular structures, and they proven to be more computationally complex.

For each test, three measurements were taken. The mean value of execution time (Time) and the number of executed rules (Nrules) are presented for each test in Table 1. As it can be seen, refactoring in AGG is quite slow, which can be explained by the fact that AGG is implemented in Java and is

Table 1. Some of the conducted refactoring tests and their performance: **Time** is average test completion time in AGG (taken across 3 measurements, standard deviation of 6%), and **Nrules** is the number of rules applied.

| | |
|---|---|
| 1. *ECCC-chain:* a sequence of an E-arc followed by C-arcs. The dependency of arc conditions on preceding EC – actions is taken into account (shown as dot-dash arc: - · - · - > ). **Time = 11.2 sec; Nrules = 13.**  | 6. *Irregular structure*[1] #1. **Time = 27 sec; Nrules = 35.**  |
| 2. *Dotted_line:* a sequence of an E-arc, followed by intermittent *C*- and *T*-arcs. **Time = 14.5 sec; Nrules = 33**  | 7. *c_Irregular1:* irregular structure that includes a state with one C-arc and one E-arc. **Time = 23.0 sec; Nrules = 32.**  |
| 3. *Common_chain* of two C-arcs used to relay signal from two different E-arcs. **Time = 13.5 sec; Nrules = 29.**  | 8. *Square* with closed loop of E-arcs. **Time = 23.8 sec; Nrules = 35.**  |
| 4. *Tuft* of an E-arc followed by branching C-arcs. **Time = 10.4 sec; Nrules = 14.**  | 9. *Double_Square* with two closed loops of E-arcs sharing one such arc. **Time = 35.2 sec; Nrules = 49.**  |
| 5. *Each_to_Each:* consists of "states' layers", where each state of a preceding layer is connected to each state of the following layer, first by E-arc and then by C-arcs. **Time = 21.5 sec; Nrules = 28.**  | 10. *Linear2* – Linear C-arcs chain with E-arcs bridges; **Time = 70.6 sec; Nrules = 117.**  |
| Legend: ⟶ - E-arc, ⟶ - T-arc; - - - -> - C-arc; | |

intended for research use. Re-implementation of the transformation rules in C can bring 20-times speedup, which will make application of the most of the rules practical as a part of an IDE.

To check the performance dependency on the dimension of ECC, sets of tests with different dimensions were generated. In Figure 21, results of two such tests are presented: *Linear*, similar to that of Figure 2, and *Linear2* from Table 1 (#10). For *Linear*, refactoring time seems to be a linear function of ECC size, and for *Linear2* the dependency seems to be polynomial.

The termination of graph rewriting is undecidable in general (but the termination can be preserved if the transformation rules meet certain criteria) [22]. Thus, termination of the refactoring problem is, in general, undecidable too. In practical terms, there can be a situation, when refactoring takes long time and it is impossible to predict upfront whether it will ever terminate. However, all our tests do terminate.

We have checked the compliance of our refactoring system with the termination criterion using *AGG*. It turned out, however, that the criterion is not satisfied. Indeed, an infinite derivation in case of a cyclic *CT*-network of an ECC is possible, but as we have mentioned above, such ECCs are beyond the scope of our consideration. But it should be noted that the *AGG* analysis does not take into account deep off-stage dependencies among attributes expressed in user-defined Java-classes.

On the other hand, our graph transformation approach is a particular case of graph rewriting systems, which, in turn,

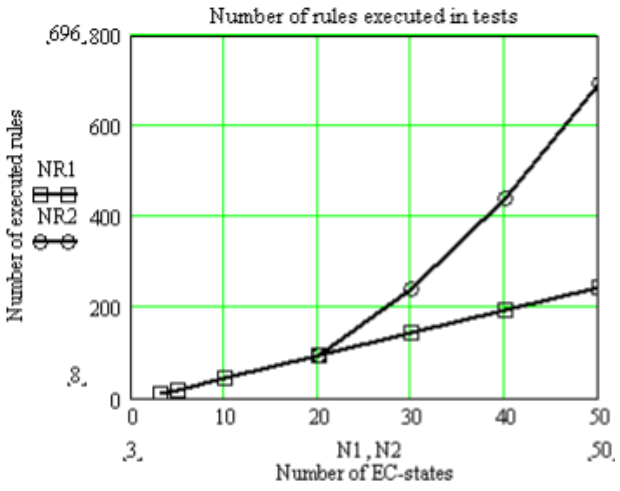



Figure 21.Number of executed refactoring rules for the tests *Linear* (NR1) and *Linear2* (NR2) as a function of ECC states' number.

belong to the general class of systems based on production rules. Many problems of graph rewriting systems are known inherited problems of such systems, but nevertheless, production rule-based reasoning is very popular, for example in various knowledge representation systems.

*Completeness* of the system of rules means that it would be possible to guarantee achieving a particular refactoring goal (say absence of deadlocks) with the developed system of rules in any source ECC. Our system of refactoring rules is created empirically. We have not attempted so far to prove formally, whether our system is complete. However, all conducted tests meet the completeness property.

The practical role of *confluence* is to ensure same result if the rules' are applied in different order. Here we rely on the result of [21] stating that typed attributed graph transformation system is locally confluent if all *critical pairs* of rules are confluent. By definition from [9, 21] a critical pair of rules (p1, p2) exists if the application of p1 disables that of p2 or, vice versa. The AGG system can do the critical pair analysis for a given system of rules. The set of discovered critical pairs represents precisely all potential conflicts. Using AGG we have discovered and fixed some mistakes in the rules of our refactoring system. However, the process of critical pair analysis is computationally complex and slow in the current AGG implementation. To test confluence we ran each of our tests several times and always obtained same results, thus confirming the property experimentally.

## X. Conclusion

In this paper a graph-transformation approach to refactoring of Execution Control Charts of basic Function Blocks is presented. This approach is extended to the correction of ECCs by removal of deadlock states.

Future work in this direction is envisaged along the following lines:

- Properties of the developed system of rules will be further investigated, in particular their completeness and confluence.
- The system of refactoring rules can be further extended to take into account particular execution semantics of function blocks.
- Refactoring can be applied not only to basic FBs, but also to FB networks. Possible ideas may include substitution of an arbitrary sub-network of FBs by the equivalent composite FB, or substitution of multiple connections between FBs by adapter connections.

The IEC 61499 standard presents a novel visual programming approach to the design of automation systems and refactoring undoubtedly can be a very useful feature of the corresponding engineering tools. Refactoring can be especially important in industrial automation applications which are facing problems of software lifecycle adaptability, same as business software applications, but a lot more sensitive to software faults. Application of the proposed refactoring technique can help to avoid introduction of new errors during the process of software modifications.

## XI. Acknowledgement

The authors thank nxtControl GmbH for providing the function block editing tool nxtStudio used for preparation of the test example in Figures 1 and 18.

## XII. REFERENCES


1. Function blocks for industrial-process measurement and control systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
2. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999
3. Sendall, S., Kozaczynski, W., "Model transformation: The heart and soul of model-driven software development", IEEE Software, Special Issue on Model-Driven Software Development. - 2003.20 (5), p.42-45
4. Object Management Group (OMG) Web-site http:// www.omg.org
5. Ledeczi, *et al.*, "Composing Domain-Specific Design Environments", Computer, Nov. 2001, pp. 44-51
6. Thramboulidis K., "Model-integrated mechatronics - Toward a New Paradigm in the Development of Manufacturing Systems", IEEE Trans on Industrial Informatics, 1(1), 2005
7. Ehrig, H., *et al.*, "Handbook of Graph Grammars and Computing by Graph Transformations" - World Scientific, 1999- v.1
8. Grunske, L., *et al.*, "Graph Transformation for Practical Model Driven Software Engineering", in: Model-driven Software Development. - Springer, 2005. p. 91-118
9. Mens, T., "On the Use of Graph Transformations for Model Refactoring", Lecture Notes in Computer Science: Generative and Transformational Techniques in Software Engineering - 2006. vol. 4143. - p.219-257
10. Dubinin, V., Vyatkin V., "Building of search and transformation systems for support of component-based industrial automation systems design", Proc. of IEEE Conf. AIS'05, Divnomorskoe, Sept, 2005, vol.2, p.30-35
11. Zoitl, A., *et al.* "The Past, Present, and Future of IEC 61499", Lecture Notes In Artificial Intelligence; Vol. 4659, Proceedings of the 3rd International Conference on Industrial Applications of Holonic and Multi-Agent Systems: Holonic and Multi-Agent Systems for Manufacturing, Regensburg, Germany, 2008
12. Vyatkin, V., Hanisch, H.-M. „Verification of Distributed Control Systems in Intelligent Manufacturing", Journal of International Manufacturing, 14, (1), p. 123-136, 2003
13. Gengic, G., "On Formal Methods in Development of Control Logic Using IEC 61499", Doctoral thesis, Chalmers University of Technology, 2009, ISBN 978-91-7385-241-8
14. Function Block Development Kit, Online: www.holobloc.org
15. Taenzer, G., "AGG: A Tool Environment for Algebraic Graph Transformation", Lecture Notes in Computer Science vol.1779, Springer, 2000, p.481-490
16. AGG Web-site http: //tfs.cs.tu-berlin.de/agg
17. Dubinin, V., Vyatkin, V., "Towards a Formal Semantics of IEC 61499 Function Blocks", 4th IEEE Conference on Industrial Informatics (INDIN'06). - Singapore, 2006, p.6-11
18. Vyatkin, V., Dubinin, V., "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC 61499", 5th IEEE Conf. on Industrial Informatics (INDIN'07), Vienna, 2007- pp.1183-1188
19. Sünder, C. *et al.,* "Usability and Interoperability of IEC 61499 based distributed automation systems", 4th IEEE Conference on Industrial Informatics (INDIN'06). - Singapore, 2006
20. Ehrig H., Prange U., Taenzer G., "Fundamental theory for typed attributed graph transformation", In: Graph Transformation: Second International Conference, ICGT'04, Lecture Notes in Computer Science, Springer-Verlag, 2004, pp.161-177
21. Heckel,R., Malte Küster J., Taentzer G., Confluence of Typed Attributed Graph Transformation Systems, in: Proc. ICGT 2002. Volume 2505 of LNCS, Springer, 2005
22. Ehrig H., Ehrig K., Lara J., Taentzer G., Varro D., and Varro-Gyapay S., „Termination Criteria for Model Transformation", LNCS, vol. 3442/2005, Berlin-Heidelberg
23. 4DIAC-IDE web-site: http://www.fordiac.org/9.0.html
24. FBench website: http://www.ece.auckland.ac.nz/~vyatkin/fbench/


**Valeriy Vyatkin** -- (SM'04) graduated with a Diploma degree in applied mathematics in 1988, received the the Ph.D. degree in 1992 and Dr. Sci. degree in 1998 from Taganrog State University of Radio Engineering (TSURE), Taganrog, Russia, and the Dr. Eng. Degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999. Currently, he is Senior Lecturer at the Department of Electrical and Computer Engineering at the University of Auckland, Auckland, New Zealand. His previous faculty positions were with Martin Luther University of Halle-Wittenberg in Germany (Assistant Professor, 1999–2004), and with TSURE (Senior Lecturer, Professor, 1991–2002). He is Program Director of Software Engineering and the Head of the infoMechatronics and IndusTRial Automation lab (MITRA). His research interests are in the area of industrial informatics, including software engineering for industrial automation systems, distributed software architectures (e.g. IEC 61499), multi-agent systems, methods of formal validation of industrial automation systems, and theoretical algorithms for improving their performance.

**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 re-searcher 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 and 2006 he has been awarded DAAD-grants to work as a guest scientist at Martin-Luther-University Halle-Wittenberg, Germany. His research interests include formal methods for specification, verification, synthesis and implementation of distributed and discrete event systems.