Reconstruction of Function Block Logic using Metaheuristic Algorithm

Daniil Chivilikhin, *Member, IEEE*, Anatoly Shalyto, *Member, IEEE*, Sandeep Patil, *Student Member, IEEE* and Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—An approach for automatic reconstruction of automation logic from execution scenarios using a metaheuristic algorithm is proposed. IEC 61499 basic function blocks are chosen as implementation language and reconstruction of Execution Control Charts for basic function blocks is addressed. The synthesis method is based on a metaheuristic algorithm that combines ideas from ant colony optimization and evolutionary computation. Execution scenarios can be recorded from testing legacy software solutions. At this stage results are only limited to generation of basic function blocks having only Boolean input/output variables.**

***Keywords*—Automatic model synthesis, industrial automation software, evolutionary computation, control system synthesis.**

I. Introduction

THE IEC 61499[1] standard [1] defines an open architecture for distributed control and automation systems. The elementary component of IEC 61499 is a *function block* (FB). All FBs are characterized by an *interface*, which defines used input/output events and input/output variables. *Basic* FBs are represented by event-driven *Execution Control Charts* (ECCs), which are Moore finite-state machines (FSMs). *Composite* FBs are defined by a network of other FBs, either basic or composite.

Migration from legacy automation systems based on Programmable Logic Controllers (PLCs) to the new generation IEC 61499 systems has been actively addressed by the research community. In addition to the widely claimed flexibility and distributability of IEC 61499 applications, the state machine based programming of IEC 61499 FBs offers much better readability and maintainability of software, though some issues regarding industrial acceptance [2] and different execution semantics [3], [4] persist. However, most works on migration assume that PLC code is available and propose methods of generating an equivalent FB application in IEC 61499. This would not help in frequently encountered situations when the source code is no longer available. It also applies to long-supported projects in IEC 61499 – source code can be lost over time.

An ideal solution would be to put the legacy system into a test environment and record traces of its behavior, after which a software tool might be used to reconstruct the code automatically. There are quite a few approaches to test case generation for industrial automation systems [5], [6], however most of them also require access to source code. For generating test cases in the abscense of source code we need to turn our attention to black-box testing (nothing is known about the system under test except its reactions to input signals).

This paper attempts to make a first step to automate the development process of IEC 61499 applications. The contribution of the paper is an approach that, under several simplifications, is able to infer an ECC of a basic FB from examples of its behavior – sequences of input/output variable value sets called *execution scenarios*. The approach is able to infer both the transition diagram and the algorithms associated with ECC states. This result opens a perspective that one day an engineer, instead of designing an ECC himself, could supply several test cases to a procedure that would automatically generate the ECC.

In this paper ECCs are inferred using a metaheuristic algorithm MuACO [7] based on ant colony optimization [8] and evolutionary computation [9]. The proposed approach falls into the *search-based software engineering* framework [10], [11], where search-based optimization techniques, such as ant colony optimization or evolutionary computation, are used for solving various problems arising in the software engineering domain. Preliminary results were published in conference proceedings [12], but this extended version has the following unique contributions: (1) handling input/output events – earlier we assumed that only the REQ input event and CNF output event are used and (2) handling multiple output actions per ECC state instead of only one.

The rest of this paper is structured as follows. Section II reviews some related work. In Section III we provide a more formal statement of the problem solved in this paper. Section IV describes the proposed approach. Experimental results are described and discussed in Section V and Section VI concludes.

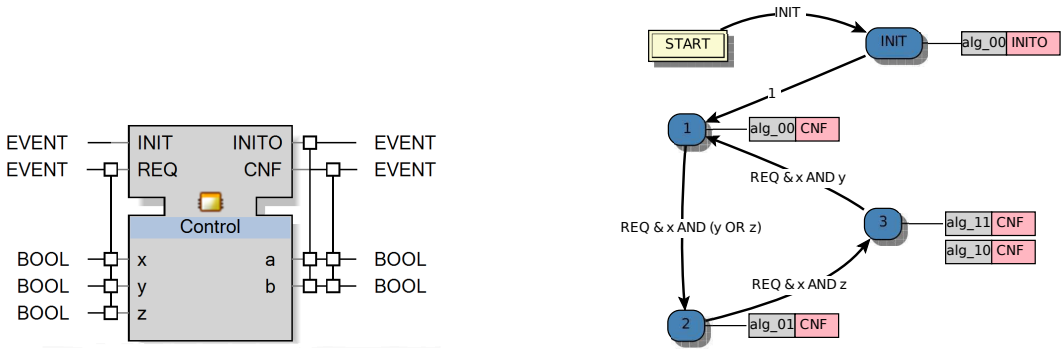
D. Chivilikhin and A. Shalyto are with the Computer Technologies Laboratory, ITMO University, St. Petersburg 197101, Russia (email: chivdan@rain.ifmo.ru, shalyto@mail.ifmo.ru).

S. Patil is with the Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, Luleå 97187, Sweden (email: sandeep.patil@ltu.se).

V. Vyatkin is with the Department of Electrical Engineering and Automation, Aalto University, Espoo 02150, Finland, and also with the Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, Luleå 97187, Sweden (e-mail: vyatkin@ieee.org).

[1]V. Vyatkin, IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design, 2nd Edition, ISA 2012

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2017.2710224, IEEE Transactions on Industrial Informatics



(a) Function block interface example (b) Execution control chart example

Fig. 1. Examples of a function block interface (left) and execution control chart (right)

II. Related Work

There has been a substantial body of publications on migration from PLCs to IEC 61499, for example [13], [14], [15], [16], but all such methods assume availability of source code. We emphasize that the approach developed in this paper does not use source code in any way.

This work is based on the recent progress in inferring various kinds of finite-state machine models. The most closely related work is devoted to inferring FSMs for controlling an unmanned aircraft [17]. In this paper FSMs are inferred from execution scenarios recorded by a human pilot using a flight simulator. The inferred FSM is able to reproduce a maneuver the pilot recorded in scenarios. A scenario element consists of real values: a set of flight parameters (e.g. altitude, airspeed) and a set of aircraft control parameters (e.g. elevator, ailerons). Real input variables are converted to Boolean variables using human-designed predicates. However, this method is not applicable to the problem addressed in this paper since it (1) does not support general form Boolean formulas on transitions, (2) uses a different state machine execution semantics than the one used in IEC 61499 and (3) does not allow to have several output actions per one state.

Another group of methods address the problem of inferring behavior models of software. In [18] an algorithm for inferring finite-state models of software from traces has been proposed. The algorithm also takes into account temporal constraints expressed in Linear Temporal Logic. Models inferred in that paper can be used for verification and testing, but cannot substitute software they were inferred from.

Finally, a powerful method exists for inferring extended finite-state machines (EFSMs) from scenarios [19]. The method is based on translating the problem of EFSM inference to Boolean satisfiability (SAT) and using state-of-the-art SAT-solvers. General form Boolean formulas on transitions are supported, however the efficiency of the method greatly diminishes when the number of input variables and input data length increase. Furthermore, this method is inapplicable for the type of input data available in the problem solved in this paper.

In conclusion of this section we can note that all state machine inference methods are specialized for the concrete type of inferred machine (EFSMs, finite-state transducers, etc.) and available type of training data. To the best of our knowledge, no attempt has yet been made to infer IEC 61499 function block logic from any type of data.

III. Problem Statement

A. IEC 61499 Standard

IEC 61499 applications are designed in the form of a network of interconnected FBs. Each FB has an *interface* defining input/output events and variables. Variables can be, for example, Boolean, integer, or real, and can be associated with input and output events. Such associations mean that upon receiving an event the FB requests the latest values of associated variables.

Basic FBs are represented by Moore finite-state machines called Execution Control Charts (ECCs). An ECC is comprised of a set of *states* connected with *transitions*. In the beginning, the ECC is in the *initial state*. When an input event is received, the ECC switches to another state if one of the transitions is triggered. This happens if the *guard condition* (Boolean formula over input/output/internal variables and constants) of a transition is satisfied. Transitions are checked in the order they are recorded in the source file of the FB; the first transition for which the guard condition is satisfied is triggered.

A state can have a number of associated *output actions*, each action consists of an *algorithm* execution and *output event* generation. Algorithms are commonly implemented using the *Structured Text* language and used, for example, for setting output variable values. An example of an FB interface is shown in Fig. 1a, associations of events with variables are depicted by vertical lines. An example of an ECC is shown in Fig. 1b. Note that the "1" guard condition on the transition from state "Init" to state "1" denotes "True". As defined in the IEC 61499 standard, "1" denotes "a transition condition with no associated event and a guard condition that is always TRUE". Output algorithms are denoted "alg_00"–"alg_11": for example, "alg_01" is implemented as $a = 0$, $b = 1$.

Note that state "3" has two associated output actions with algorithms "alg_11" and "alg_10". It might seem that the first action "alg_11" is redundant, since after the execution of the second algorithm output variable b will be set to 0. However,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2017.2710224, IEEE Transactions on Industrial Informatics

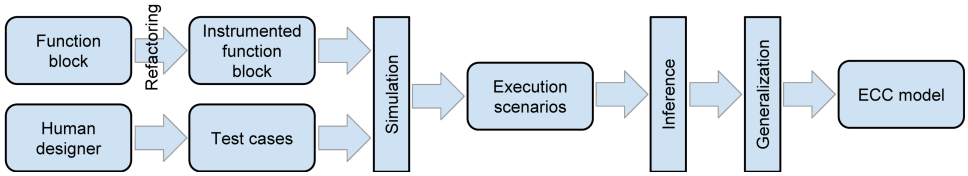


Fig. 2. Proposed approach scheme

it is not the case: since this action also generates a "CNF" output event, other FBs may be triggered by b being set to 1 by the first algorithm.

B. Definitions

In this work we will consider a simplified model of the ECC, in which (1) all input and output variables are Boolean and (2) guard conditions depend on input variables only. Assuming that, an execution control chart is defined as an eight-tuple $\langle Y, EI, X, EO, Z, y_0, \phi, \lambda \rangle$, where Y is a finite set of states, EI is a set of input events, X is a set of Boolean input variables, EO is a set of output events, Z is a set of Boolean output variables, $y_0 \in Y$ is the initial state, $\phi \colon Y \times EI \times \{0,1\}^{|X|} \to Y$ is the transitions relation and $\lambda \colon Y \to \{\Omega_y^i \times EO \mid \Omega_y^i \colon \{0,1\}^{|Z|} \to \{0,1\}^{|Z|}\}_i$ is the outputs relation.

Without loss of generality we can assume that the initial state y_0 is always state 0. The outputs relation defines that each state is associated with a list of output actions, each action consisting of an algorithm and an output event. In our simplified case algorithms are functions over output variables that transform a Boolean string to another Boolean string.

In this work we define an *execution scenario* s as a sequence of execution scenario elements s_i, where each element consists of an input event e^{in}, a set of input variable values χ, and a list of output actions $O_0 \dots O_{N_{s_i}-1}$ (N_{s_i} denotes the number of output actions in s_i). Each output action O_t is described by a set of output variable values ζ^t and an output event e_t^{out}. For example, if there are three input and two output variables,

$$\langle REQ, 000, \{(00, CNF)\}\rangle, \langle REQ, 001, \{(01, CNF), (10, CNF)\}\rangle, \langle REQ, 101, \{(11, CNF)\}\rangle$$

is an execution scenario.

We formulate the problem solved in this paper in the following way: design a method that, given a basic FB with known interface but unknown ECC, produces an ECC that complies with supplied execution scenarios.

IV. Proposed Approach

The overall scheme of the proposed approach is shown on the diagram in Fig. 2. The input FB is first refactored in order to allow execution scenarios recording [12]. A human designer supplies test cases. The FB application with the refactored FB is run on test cases, execution scenarios are recorded. Scenarios are fed to the proposed ECC inference algorithm. Finally, the inferred ECC model is generalized.

The model inference algorithm we use is based on the MuACO algorithm [7]. This algorithm is *metaheuristic*; such algorithms can be used for finding good solutions of hard optimization problems in reasonable time. In general, all metaheuristics perform a guided search in the *search space* (set of all feasible solutions) of the optimization problem. The solution considered by the algorithm at any point in time is called a *candidate solution* or an *individual*. Such algorithms are commonly used when the search space of an optimization problem is too large to be searched completely. The problem considered in this paper is closely related to the problem of learning deterministic finite automata, which has been proven by Gold to be NP-complete [20]. Therefore, metaheuristic algorithm is used.

A. Execution Control Chart Models

In this paper we use two ECC models: the first one which we call *simple* is used during inference, and the second *full* model is used during ECC generalization. The models differ in the way the transitions relation is represented. Below we first describe the full model and then define the simple model on its basis.

1) Full ECC Model: The easiest way to represent the transitions relation ϕ of an ECC in an individual of a metaheuristic algorithm is to store a $|EI| \times 2^{|X|}$ table for each state: a transition array of $2^{|X|}$ elements for each input event. However, due to the possibly large number of input variables this naive approach is infeasible. A better way to represent the transitions relation is the *reduced tables* approach [21]. In this approach it is assumed that not all input variables are essential for determining the appropriate transition in each state. Variables that are necessary are called *significant*, all other variables are called *insignificant*. Indeed, it is often the case that, though an FB has ten input variables, two or three variables are enough for making the right transition choice. Each state in the reduced tables approach has an associated *significance mask* m, which contains a Boolean significance variable for each input variable. If for some state $m_i = true$, then the input variable x_i is significant in this state.

However, simply adopting the reduced tables approach is not enough for modeling ECCs. The reason is that this approach only allows to represent rather simple Boolean formulas which do not use parentheses or disjunction, e.g. $x_1 \wedge \neg x_2 \wedge x_3$. Boolean formulas on ECC transitions are often more complicated, e.g. $x_1 \wedge (\neg x_2 \vee \neg x_3)$. To represent such formulas we will use the fact that any Boolean formula can be represented in Disjunctive Normal Form (DNF): $(\dots \wedge \dots \wedge \dots) \vee \dots \vee (\dots \wedge \dots \wedge \dots)$. Each state will have a set of associated *transition*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2017.2710224, IEEE Transactions on Industrial Informatics

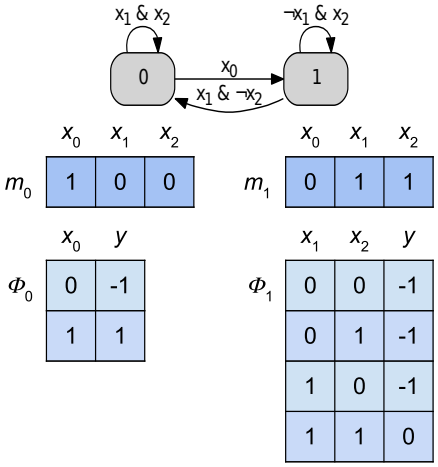


Fig. 3. An example of an ECC model and the representation of state 0. The "-1" entries mean that the corresponding transitions are not present in the transition group.

groups for each input event, where each transition group is a reduced transition table with its own significance mask. For example, formula $x_1 \wedge (\neg x_2 \vee \neg x_3)$, which is equivalent to $(x_1 \wedge \neg x_2) \vee (x_1 \wedge \neg x_3)$, can be represented with two transition groups with variables (x_1, x_2) and (x_1, x_3) being significant in the first and second group respectively.

Summing up, our full ECC model is represented as a set of states Y, where each state y has a set of transition groups T_y for each input event. Each transition group $t \in T_y$ has an associated Boolean array called the input variable significance mask m_t and a transition table Φ_t of $1 \times 2^{\text{sum}(m_t)}$ elements, where $\text{sum}(m_t)$ is the number of elements in m_t that are *true*. Each j-th element of Φ_t stores the new state for the corresponding transition. For example, if there are four significant input variables x_0–x_3, Φ_t^5 stores the new state that the ECC has to change to when $(x_0, x_1, x_2, x_3) = (0, 1, 0, 1)$ since $0101 = 5$ in the binary system. The model also includes the number K_y of output actions for each state y of the ECC. Examples of an ECC model and its representation are shown in Fig. 3: the ECC model is the state diagram on top and the representation of transitions from state 0 is at the bottom.

2) Simple ECC model: In the simple ECC model we assume that all input variables are significant in all states. However, we do not store the transition tables for all possible combinations of input variable values. Instead, only the ones present in scenarios are considered. This approach reduces the size of the search space.

The simple ECC model is sufficient for the inference step, since during inference it is impossible to encounter a combination of input variables that is not present in scenarios. However, it is insufficient for consequent use of the generated ECC, since such an ECC would not generalize to unseen data. Therefore, we also use the full model.

3) Output relation representation: Both ECC models use the same output relation representation. Since here all output variables are binary, an algorithm can be represented as a string of length $|Z|$ over the alphabet {"0", "1", "x"}. Let a_{yt} be the t-th algorithm associated with state y. The i-th character of the algorithm a_{yt}^i is associated with the i-th output variable. Algorithms have the following semantics:

- $a_{yt}^i =$ "0": set $z^i \leftarrow 0$;
- $a_{yt}^i =$ "1": set $z^i \leftarrow 1$;
- $a_{yt}^i =$ "x": preserve current value of z^i.

Instead of storing algorithms and output events in the model and evolving them simultaneously with the ECC transitions, we deduce them from execution scenarios before fitness evaluation using a state labeling algorithm based on the same idea as the original one from [22].

B. ECC Inference Algorithm

For ECC model inference we use the parallel version of the MuACO algorithm [23], [7]. The algorithm starts with a randomly generated initial solution and explores the search space using *mutation operators*, which make rather small changes to the ECC. The degree to which a candidate solution complies with execution scenarios is evaluated using a so-called *fitness function*. Before computing the fitness function value of a candidate solution, the algorithms for each state are determined using state labeling.

C. Mutation Operators

The following two mutation operators were used.

Change transition end state. First we select a random state with transitions that have been used during fitness evalution (*used transitions*). If the model does not have used transitions, an arbitrary state is selected. Second, a random (used) transition is selected and the state y it leads to is changed to another state selected uniformly at random from $Y \setminus \{y\}$.

Change number of output actions in a state. This mutation operator changes the number of output actions K_y for a randomly selected state y to a new value situated between one and the maximum length of a sequence of output actions present in scenarios (inclusively).

D. State Labeling Algorithm

For each state y and output action O_t, $t \in [0 \dots K_y - 1]$ a list of pairs of strings P_y^t is stored. The labeling algorithm consecutively processes all execution scenarios. Before processing each scenario the ECC is in its initial state. Input events and input variable sets of scenarios are fed to the ECC one by one.

Consider processing the k-th scenario element ($k > 0$) when the current state is y. The ECC makes the transition induced by the set of input variables $s_k.\chi$, changing the current state y to a new value y_{new}.

First we add the pair $\langle s_{k-1}.\zeta^{K_y - 1}, s_k.\zeta^0 \rangle$ to $P_{y_{\text{new}}}^0$. This deals with moving from the last output action of the previous scenario element to the current scenario element.

Second, for each two consecutive output actions O_t and O_{t+1} ($t > 0$) in element s_k the pair $\langle s_k.\zeta^t, s_k.\zeta^{t+1} \rangle$ is added to the list $P_{y_{\text{new}}}^t$. This deals with the fact that a scenario element may have more than one output action.

After all scenarios have been processed, algorithms for all states are determined according to Algorithm 1. Labels

Algorithm 1 State labeling algorithm

Require: List of pairs P_y^t for each y and each t
1: **for all** $y \in Y$ **do**
2: **for all** $t \in [0 \dots K_y - 1]$ **do**
3: $a_{yt} \leftarrow$ new char$[|Z|]$
4: **for** $i = 0$ to $|Z| - 1$ **do**
5: $d_0 \leftarrow 0, d_1 \leftarrow 0, d_x \leftarrow 0$
6: **for all** $(l, r) \in P_y^t$ **do**
7: $d_{r_i} \leftarrow d_{r_i} + 1$
8: **if** $l_i = r_i$ **then**
9: $d_x \leftarrow d_x + 1$
10: **end if**
11: **end for**
12: $a_{yt}^i \leftarrow \arg\max \{\langle$"0", $d_0\rangle; \langle$"1", $d_1\rangle; \langle$"x", $d_x\rangle\}$
13: **end for**
14: **end for**
15: **end for**

Algorithm 2 Fitness function for a single scenario

Require: M – ECC model, s – scenario
1: $y \leftarrow 0$, $n_{\text{sc}} \leftarrow 0$, $n_{\text{fe}} \leftarrow -1$
2: $z \leftarrow a_y.\texttt{apply}(0^{|Z|})$
3: $\Delta^{\text{var}} \leftarrow 0, \Delta^{\text{events}} \leftarrow 0$
4: **for** $i = 0$ to $|s| - 1$ **do**
5: $y_{\text{next}} \leftarrow M_y.\texttt{nextState}(s_i.e^{\text{in}}, s_i.\chi)$
6: **if** $y_{\text{next}} \neq -1$ **then**
7: $y \leftarrow y_{\text{next}}$, $z \leftarrow a_y.\texttt{apply}(z)$
8: $n_{\text{sc}} \leftarrow n_{\text{sc}} + 1$
9: $e^{\text{out}} \leftarrow M_y.e^{\text{out}}$
10: $\delta^{\text{var}} \leftarrow \frac{1}{|z|}\Delta_H(s_i.\zeta, z)$, $\delta^{\text{events}} \leftarrow I(e^{\text{out}} \neq s_i.e^{\text{out}})$
11: **if** $(n_{\text{fe}} = -1) \wedge (\delta^{\text{var}} > 0 \vee \delta^{\text{events}} > 0)$ **then**
12: $n_{\text{fe}} \leftarrow i$
13: **end if**
14: $\Delta^{\text{var}} \leftarrow \Delta^{\text{var}} + \delta^{\text{var}}$, $\Delta^{\text{events}} \leftarrow \Delta^{\text{events}} + \delta^{\text{events}}$
15: **end if**
16: **end for**
17: $F_{\text{err}} \leftarrow 1 - \frac{\Delta^{\text{var}}}{2\max(\Delta^{\text{var}}, |s|)} - \frac{\Delta^{\text{events}}}{2\max(\Delta^{\text{events}}, |s|)}$
18: $F_{\text{fe}} \leftarrow \frac{n_{\text{fe}}}{|s|-1}$, $F_{\text{sc}} \leftarrow 1 - \frac{n_{\text{sc}}}{|s|}$
19: **return** $c_1 F_{\text{ed}} + c_2 F_{\text{fe}} + c_3 F_{\text{sc}}$

(algorithms) for each state y and each action position t are determined separately. In line 3 the string a_{yt} representing the algorithm for state y in position t is initialized. Then in line 4 we iterate over all characters in the algorithm string. Each such character is also determined separately. In line 5 we initialize variables d_0, d_1 and d_x. The first two are used for storing how many times the second symbol of a pair is 0 or 1, variable d_x is incremented if symbols in a pair are equal. The i-th character of the algorithm is selected in line 12 as an argument of the maximum value of a decision relation. Output events are determined by a similar procedure.

E. Fitness Function

The fitness function we used consists of three components:

$$F = c_1 F_{\text{err}} + c_2 F_{\text{fe}} + c_3 F_{\text{sc}},$$

where F_{err} is based on the cumulative *errors* the ECC makes, F_{fe} is based on the position of the *first error* made by the model, F_{sc} is the number of *state changes*, and c_1–c_3 are constants. The process of calculating a fitness function value for one scenario is described in Algorithm 2. Evaluation of several scenarios is done by averaging the corresponding fitness values for individual scenarios. In this pseudocode we omit implementation details and assume for simplicity that each state and each scenario element have exactly one associated output action.

In the beginning y is the initial state 0. Variable n_{sc} is used for counting the number of state changes made by the ECC. Variable n_{fe} stores the index of the scenario element on which the ECC makes the first error in output variable values or output event. Variable z holds the current values of output variables. Expression $a.\texttt{apply}(z)$ (lines 2, 7) denotes the application of the algorithm a to output variable values z. For example, "x0x1".$\texttt{apply}(0110) = 0011$.

The "for" loop iterates over scenario elements (lines 4–16). For each element the next state is determined using the `nextState` function based on the current state y, input event $s_i.e^{\text{in}}$ and input variable values $s_i.\chi$ (line 5). If the transition exists ($y_{\text{next}} \neq -1$), then the current state is updated, the new algorithm is applied to the current output variables, and the number of state changes is increased. In line 10 the errors δ^{var} and δ^{events} in output variable values and output events are calculated. The Δ_H function computes the Hamming distance between two strings and I is an indicator function that equals one iff its Boolean argument is true. Then in lines 11–13 we check if the first error in output variable values and/or output events occurred at the current scenario element.

In lines 17–18 fitness function component values are determined and in line 19 the final fitness function value is calculated. Coefficients c_1–c_3 are selected such that the maximum value of the fitness function is 1.0001 (this exact value cannot be reached in practice unless a correct ECC makes zero state changes, which is impossible). In experiments reported in this paper $c_1 = 0.1$, $c_2 = 0.9$, $c_3 = 0.0001$.

Due to the large total length of scenarios, evaluating all candidate solutions using full scenarios is very computationally expensive. Therefore we used a hierarchy of shortened sets of scenarios. Scenario shortening was done by finding all subsequences of equal scenario elements and collapsing each such sequence down to a maximum of n_{scale} elements. A candidate solution is first evaluated using the shortest set of scenarios. If it passes all scenarios and has a fitness value greater than or equal to 1, it is evaluated using a larger set of scenarios. This approach was inspired by [24].

F. Model Generalization

To generalize the inferred simple ECC model to a full model the following post-processing procedure is performed after a perfect solution is found. A greedy algorithm is used: at each step the model is modified, the change is retained if the fitness function value of the modified model value did not decrease.

First we try to delete each transition from each state. Second, we try to delete each significant input variable of each transition group. First the selected variable is made insignificant: $m_i \leftarrow false$. The size of the transition table is halved. When a variable is made insignificant, we have to choose which of the two transitions to preserve – the one with $x_i = true$ or with $x_i = false$. We first try to keep the transition with $x_i = false$. If this makes the ECC incorrect, we consequently attempt to leave the transition with $x_i = true$. The process is repeated until no fitness non-decreasing changes can be made. Finally, an attempt is made to generalize output algorithms: each algorithm element is replaced with "x", the change is retained if it does not decrease the fitness value.

V. Experiments

The purpose of the experiments was to demonstrate the principal viability and applicability of the proposed approach to basic FB reconstruction from behavior examples without the use of source code. Some of the default parameter values for the parallel MuACO algorithm from [23] were altered: $N_{\text{stag}} = 100$, $N_{\text{mut}} = 10$. Selected values are better suited to the large search space of the ECC inference problem. In the fitness function we used two shortened sets of scenarios with $n_{\text{scale}} = 1$ and $n_{\text{scale}} = 2$.

Experiments were performed for one of the basic FBs of the Pick-and-Place (PnP) manipulator project [25] (Fig. 4). The FBDK[2] development environment was used for recording execution scenarios and post-synthesis simulation, ECC plots were generated using nxtStudio[3]. This PnP implementation has two horizontal pneumatic cylinders (I and II), one vertical cylinder (III), and a suction unit (IV) for picking up work pieces. When sensors determine that a new work piece (*) appeared in one of the input trays (1, 2, 3), the PnP system retrieves the work piece and puts it on the output slider (V).

In this particular implementation logic control is performed in a centralized way by a single basic FB *CentralizedControl*. It receives signals when work pieces appear in the input trays and sends commands to other FBs that control the movement of the cylinders and the suction unit. We chose this basic FB for testing our approach because it uses only Boolean input/output variables, and implements some non-trivial logic.

The interface of the *CentralizedControl* includes the following Boolean input variables: c1Home/c1End (cylinder I is in the leftmost/rightmost position), c2Home/c2End (same for cylinder II), vcHome/vcEnd (cylinder III is in the top/bottom position), pp1/pp2/pp3 (a work piece is present in input tray 1/2/3), vac (the suction unit is on). The following Boolean output variables are used: c1Extend/c1Retract (extend/retract horizontal cylinder I), c2Extend/c2Retract (same for cylinder II), vcExtend (extend vertical cylinder III), vacuum_on/vacuum_off (turn suction unit on/off).

For comparison purposes we confront the generated ECC with the manually created one, further referred to as *original*. The original ECC of the *CentralizedControl* FB is shown in Fig. 5. It has nine states (excluding the *Start* and *Init* states) and 15 transitions. We shall note, however, that the original ECC is not anyhow used in the reconstruction process.

[2]http://www.holobloc.com/doc/fbdk

[3]http://www.nxtcontrol.com/

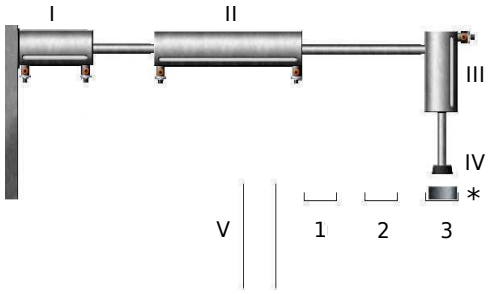


Fig. 4. Screenshot of one the Pick-and-Place manipulator system implementations

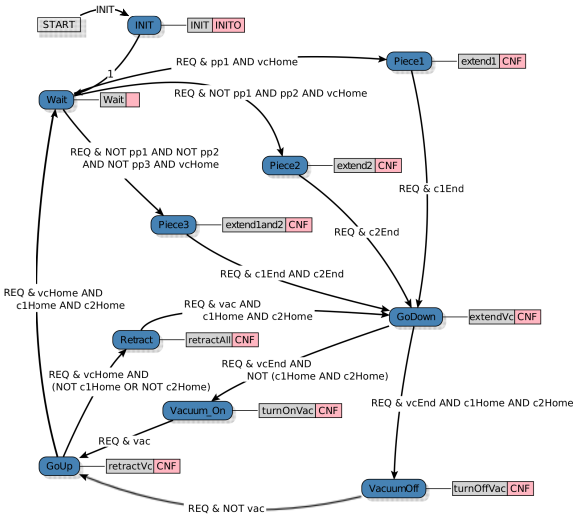


Fig. 5. Original ECC of the *CentralizedControl* FB

A. Inferring ECC Models

The experiment plan was as follows: (1) record execution scenarios, (2) use the proposed approach to infer an ECC model compliant with all scenarios and (3) test all inferred ECC models in simulation using FBDK.

Each scenario is defined by a test case, which is the order of work pieces the PnP manipulator has to process. For example, '1-2-3' denotes a test case where first the manipulator is given piece number one, then piece number two, and, finally, piece number three. We recorded 39 test scenarios based on all test cases of lengths one, two and three: from '1' to '3-3-3'. A scenario set is denoted by the number of scenarios it includes: for example, set 5 includes scenarios '1', '2', '3', '1-1', '1-2'. The length of a scenario set is calculated as the total number of elements of its scenarios, the largest scenario set has a length of 150170. It is assumed that all scenarios are independent – the original ECC is reinitialized before recording each scenario.

The number of ECC states can be estimated by calculating a set of algorithms that can be used to represent test scenarios

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2017.2710224, IEEE Transactions on Industrial Informatics

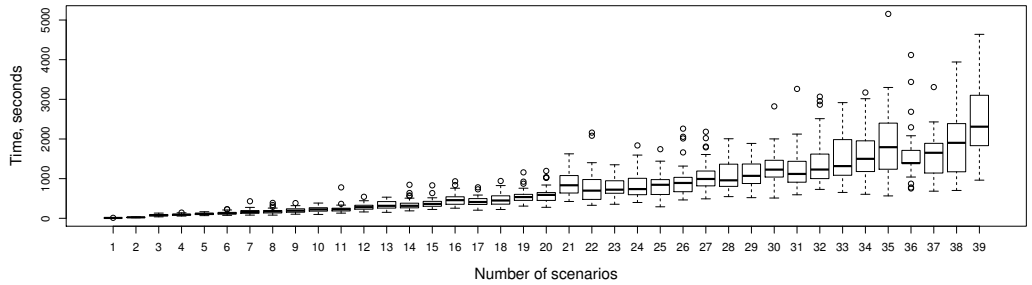


Fig. 6. Running times of the proposed algorithm in dependence from the number of test scenarios

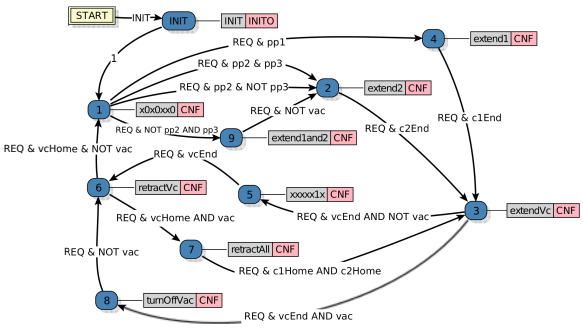


Fig. 7. One of inferred ECC models

and taking its power as an estimate of $|Y|$. Unfortunately, this is neither a lower nor an upper bound, so to find an appropriate value we have to iterate over $|Y|$. However, this is not necessary to show the viability of the method so this step was excluded from the experiment protocol.

A machine with a 64-core AMD Opteron^{TM}6378 @ 2.4 GHz processor was used. The proposed method was implemented in Java and run on 16 cores with a maximum of 32 GB of RAM. The experiment was repeated 40 times for scenario sets 1–20 and 30 times for scenario sets 21–39. Boxplots of running times in dependence from the number of test scenarios are shown in Fig. 6: each i-th boxplot corresponds to a series of experiments with the i-th scenario set. Though the span of the boxplots increases as scenario sets get larger, median running times (depicted with horizontal lines inside the boxes) increase almost linearly.

Afterwards, all solutions constructed from 39 test scenarios were tested in simulation using FBDK. Each ECC model was automatically converted to an IEC 61499 function block format file. Then it was translated to Java code using a call to a library supplied with FBDK and compiled. In simulation testing we checked that the model can correctly handle all test cases it was trained on. All constructed solutions passed this test. One of the constructed ECC models is shown in Fig. 7. Algorithms that are identical to the ones used in the original ECC are called by their names.

B. Results Analysis

The original ECC has 9 states, 15 transitions, and uses a total of 32 significant input variables in guard conditions.

TABLE I. STATISTICS ON INFERRED ECCS

n	1	2	3	7	39
Scenarios length	1271	2806	4609	15590	150170
% of covered transitions	60 %	87 %	100 %	100 %	100 %
Mean transition coverage frequency	1.6	2.2	3.93	6.06	51
# of states (min/avg/max)	7/10/10	8/10/10	9/10/10	9/10/10	9/10/10
# of transitions (min/avg/max)	10/13/16	13/18/22	13/19/24	15/21/26	18/21/24
McCabe's complexity (min/avg/max)	2/5/10	6/10/14	5/11/16	8/13/18	11/13/16
# of variables in guards (min/avg/max)	11/15/23	16/24/33	15/28/39	20/30/40	22/29/39
# of state changes (min/avg/max)	10/13/16	18/23/28	25/32/39	102/124/152	918/1162/1496
# of state changes (original)	9	18	27	99	918

To illustrate the structural complexity of ECCs, we measured McCabe's cyclomatic complexity [26], which indicates the number of linearly independent paths in an ECC. Since an ECC is a connected graph, it is calculated as the number of transitions minus the number of states plus two. McCabe's cyclomatic complexity equals 8 for the original ECC. Table I summarizes data on used test scenarios and the structure of inferred ECCs in comparison to the original one.

To evaluate how well used scenarios cover the original ECC, two notions of coverage were used: percentage of covered transitions and transition coverage frequency. The former is classical, and the latter is the mean number of times each transition is covered by the scenarios (more precisely, the number of unique scenario prefixes ending at a transition). One can notice that the classical coverage is 100 % for three scenarios already. However, this does not mean that all possible behaviors are covered.

The mean number of states is 10 for all experimental runs. In total, the complexity of generated ECCs increases as scenario sets get larger: mean number of transitions increases from 13 for one scenario to 21 for 39 scenarios, McCabe's complexity increases from 5 to 13, number of variables in guard conditions rises from 15 to 29. The table also shows the number of state changes made by the generated and original ECCs. Minimal values for generated ECCs are similar to the values for the original ECC. None of the inferred ECCs are isomorphic to the original ECC. This indicates that though all these solutions demonstrate identical *behavior* on test scenarios, their *logic* is

very different. However, characteristics of solutions generated from 39 scenarios are close to the ones of the original ECC.

These experimental results allow us to conclude that the proposed method is able to solve the addressed problem: generate an ECC that complies with given execution scenarios with no knowledge of the original ECC. Moreover, results indicate that, on average, the method scales linearly with respect to the total length of given execution scenarios.

C. Larger example: reconstructing composite FB behavior

To check the scalability of our approach we performed another experiment with a six-cylinder PnP system [27]. The controller there is implemented as a composite function block, therefore generated scenarios may have several output actions per one input event. The composite function block has 19 input and 14 output variables. Our method was able to reconstruct the behavior of the system for one test case (scenario length 19020): picking up the first work piece and delivering it to the output slider. It took the algorithm an average of 4994 seconds (min=63, max=10832) using 16 threads to construct a solution with 20 states. In comparison, for the three-cylinder PnP inferring an ECC for a similar scenario size takes only about 160 seconds. This is probably due to the target ECC having a large amount of input variables and several output actions per state – mutations that change this number are quite destructive. These results indicate that though the proposed method is in principle applicable to composite FBs, it should be specifically enhanced for this purpose to have better performance.

VI. Conclusion

We have presented an approach for reconstructing ECCs of basic IEC 61499 FBs with Boolean input/output variables and demonstrated its feasibility on two examples. In the future we plan to add support for integer and real input/output variables, as well as extend the presented approach to achieve an even wider goal – automated inference of correct-by-design function block programs. Such methods already exist for abstract models called *extended finite-state machines* (EFSMs) [28] and simpler finite-state models [18]. The basic idea is that verification is introduced into the candidate solution evaluation process. Much work has been done on formal verification of both IEC 61499 [25], [29], [30] and IEC 61131 [31], [32] software. Combining these results could bring automated inference of function block applications closer.

A standing issue of all model inference approaches including the presented one is ensuring completeness of the used set of scenarios. One approach is to have a large set of scenarios. This is supported by the proposed method since it scales well with the size of scenarios. Another argument is that the test engineer that will be responsible for test case generation should have complete knowledge of what the system should do (however, not how). Therefore he should be able to design a sufficiently complete set of tests. In addition, if the reconstructed FB demonstrates erroneous behavior (which can be discovered, e.g., by simulation testing or runtime verification), additional test cases can be derived from such behaviour. The FB should then be reconstructed again with the new test cases.

Acknowledgment

This work was financially supported by the Government of Russian Federation, Grant 074-U01, and also partially by RFBR, research project No. 14-01-00551 a. We also thank Vladimir Ulyantsev and the anonymous reviewers for useful comments.

References

[1] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[2] S. Sierla, J. Christensen, K. Koskinen, and J. Peltola, "Educational approaches for the industrial acceptance of iec 61499," in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2007, pp. 482–489.

[3] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 applications, part A: Modeling," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 136–144, 2010.

[4] ——, "On formal analysis of IEC 61499 applications, part B: Execution semantics," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 145–154, 2010.

[5] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Test case generation approach for industrial automation systems," in *5th International Conference on Automation, Robotics and Applications*. IEEE, 2011, pp. 57–62.

[6] I. Buzhinsky, V. Ulyantsev, J. Veijalainen, and V. Vyatkin, "Evolutionary approach to coverage testing of IEC 61499 function block applications," in *13th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2015, pp. 1213–1218.

[7] D. Chivilikhin and V. Ulyantsev, "MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *15th annual conference on Genetic and evolutionary computation (GECCO)*. ACM, 2013, pp. 511–518.

[8] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[9] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[10] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.

[11] M. Harman, "Software engineering meets evolutionary computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.

[12] D. Chivilikhin, A. Shalyto, S. Patil, and V. Vyatkin, "Reconstruction of function block logic using metaheuristic algorithm: Initial explorations," in *13th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2015, pp. 1239–1242.

[13] W. Dai, V. Dubinin, and V. Vyatkin, "Migration from PLC to IEC 61499 using semantic web technologies," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 3, pp. 277–291, 2014.

[14] C. Gerber, H.-M. Hanisch, and S. Ebbinghaus, "From IEC 61131 to IEC 61499 for distributed systems: a case study," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 4:1–4:8, 2008.

[15] W. Dai and V. Vyatkin, "Redesign distributed PLC control systems using IEC 61499 function blocks," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 2, pp. 390–401, 2012.

[16] M. Wenger, A. Zoitl, C. Sunder, and H. Steininger, "Transformation of IEC 61131-3 to IEC 61499 based on a model driven development approach," in *7th IEEE Conference on Industrial Informatics*. IEEE, 2009, pp. 715–720.

[17] A. Aleksandrov, S. Kazakov, A. Sergushichev, F. Tsarev, and A. Shalyto, "The use of evolutionary programming based on training examples for the generation of finite state machines for controlling objects with complex behavior," *Journal of Computer and Systems Sciences International*, vol. 52, no. 3, pp. 410–425, 2013.

[18] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2008, pp. 248–257.

[19] V. Ulyantsev and F. Tsarev, "Extended finite-state machine induction using SAT-solver," *IEEE International Conference on Machine Learning and Applications*, vol. 2, pp. 346–349, 2011.

[20] M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, no. 3, pp. 302–320, 1978.

[21] N. Polikarpova, V. Tochilin, and A. Shalyto, "Method of reduced tables for generation of automata with a large number of input variables based on genetic programming," *Journal of Computer and Systems Sciences International*, vol. 49, no. 2, pp. 265–282, 2010.

[22] S. M. Lucas and T. J. Reynolds, "Learning deterministic finite automata with a smart state labeling evolutionary algorithm," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 7, pp. 1063–1074, 2005.

[23] D. Chivilikhin and V. Ulyantsev, "Extended finite-state machine inference with parallel ant colony based algorithms," in *International Student Workshop on Bioinspired Optimization Methods and their Applications*, 2014, pp. 117–126.

[24] W. Spears and D. Gordon, "Evolving finite-state machine strategies for protecting resources," in *Foundations of Intelligent Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 1932, pp. 166–175.

[25] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *17th IEEE Conference on Emerging Technologies Factory Automation (ETFA)*. IEEE, 2012, pp. 1–7.

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[27] E. Demin, S. Patil, V. Dubinin, and V. Vyatkin, "IEC 61499 distributed control enhanced with cloud-based web-services," in *10th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2015, pp. 972–977.

[28] F. Tsarev and K. Egorov, "Finite state machine induction using genetic algorithm based on testing and model checking," in *13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO)*. ACM, 2011, pp. 759–762.

[29] L. H. Yoong and P. Roop, "Verifying IEC 61499 function blocks using esterel," *IEEE Embedded Systems Letters*, vol. 2, no. 1, pp. 1–4, 2010.

[30] S. Patil, D. Drozdov, V. Dubinin, and V. Vyatkin, "Cloud-based framework for practical model-checking of industrial automation applications," in *Technological Innovation for Cloud-Based Engineering Systems*. Springer International Publishing, 2015, vol. 450, pp. 73–81.

[31] H. Carlsson, B. Svensson, F. Danielsson, and B. Lennartson, "Methods for reliable simulation-based PLC code verification," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 267–278, 2012.

[32] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 302–310, 2012.

Daniil Chivilikhin received his B.Sc. and M.Sc. degrees in applied mathematics and informatics from ITMO University, St. Petersburg, Russia, in 2011 and 2013. In 2015 he defended his Ph.D. under the supervision of prof. Anatoly Shalyto in ITMO University, St. Petersburg, Russia. He is currently an Associate Professor at ITMO University and works in the Computer Technologies Laboratory since 2013. His research interests include finite-state models synthesis and verification, industrial informatics, constraint programming and evolutionary algorithms.

Anatoly Shalyto is a professor and a leading researcher at the department of computer technologies, faculty of IT and programming, ITMO University, St. Petersburg, Russia. His research mostly concerns automata-based programming. In particular, he introduced a methodology for automata-based programming called Switch-technology. In later years, his research is mostly dedicated to connection of automata-based programming to machine learning techniques for solving such problems as automata synthesis, testing and verification. The results of this research are currently used in a variety of Russian industrial companies. He is the author of a series of articles devoted to the problems of Computer Science and education in Russia. For his achievements in education, in 2008 he received a Russian State Government award.

Sandeep Patil (S'11) received the Bachelor's degree in computer science engineering from the CMR Institute of Technology, Bangalore, India, in 2005; the Master of computer science (software engineering) degree from the Illinois Institute of Technology, Chicago, IL, USA, in 2010; the Master of Engineering Studies (computer systems) degree from the University of Auckland, Auckland, New Zealand, in 2011; and is currently pursuing Ph.D. (Doktorand) degree in formal verification of cyber physical systems from the Dependable Communication and Computation Systems Group, Lulea University of Technology, Lulea, Sweden, supervised by chaired prof. Dr. Valeriy Vyatkin. His research interests include programming distributed industrial automation software systems using IEC 61499 standard. He is an accomplished software engineering professional with over eight years of research and development experience in systems and application software, including four years at Motorola India Pvt. Ltd., India, as a Senior Software Engineer.

Valeriy Vyatkin (M'03, SM'04) received Ph.D. degree from the State University of Radio Engineering, Taganrog, Russia, in 1992. He is on joint appointment as Chaired Professor (Ämnesföreträdare) of Dependable Computation and Communication Systems, Luleå University of Technology, Luleå, Sweden, and Professor of Information and Computer Engineering in Automation at Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar at Cambridge University, U.K., and had permanent academic appointments with the University of Auckland, Auckland, New Zealand; Martin Luther University of Halle-Wittenberg, Halle, Germany, as well as in Japan and Russia. His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; artificial intelligence, distributed architectures and multi-agent systems applied in various industry sectors, including smart grid, material handling, building management systems, data centres and reconfigurable manufacturing.

Dr. Vyatkin was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012.