Reconstruction of Function Block Logic using Metaheuristic Algorithm: Initial Explorations

Daniil Chivilikhin*, Anatoly Shalyto*, Sandeep Patil‡, and Valeriy Vyatkin*†‡
*Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia
Email: chivdan@rain.ifmo.ru, shalyto@mail.ifmo.ru
†Department of Electrical Engineering and Automation, Aalto University, Finland
Email: vyatkin@ieee.org
‡Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden
Email: sandeep.patil@ltu.se

***Abstract*—This paper presents an approach for automatic reconstruction of automation logic from execution scenarios using a metaheuristic algorithm. The IEC 61499 basic function blocks is 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 most closely related to 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.**

I. Introduction

The IEC 61499 standard[1] is a recent standard that 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 include 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. However, most of the works on migration assume that the PLC code is available and propose methods of generating an equivalent FB application in IEC 61499. This would not help in often situations when the source code is no longer available, or there are no engineers who could quickly understand it. An ideal solution would be to put the legacy system into a test environment and record traces of its behavior, after which reconstruct the code automatically.

This paper attempts to make a first step to automating the development process of IEC 61499 applications. The contribution of this 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 values sets called *execution scenarios*. The approach is able to infer both the transition diagram and the algorithms associated with ECC states.

[1]International Standard IEC61499-1: Function Blocks – Part 1 Architecture, First ed. Geneva: International Electrotechnical Commission 2005.

II. Related Work

There has been a substantial body of publications on migration from PLCs to IEC 61499, for example [1], [2], [3], but all such methods assume availability of source code. In this paper we do not require this. This work is based on the recent progress in inferring FSM models. The most closely related works are devoted to inferring FSMs for controlling an unmanned aircraft [4], [5]. 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). This method is not directly applicable to the problem solved in this paper since it (1) does not support general form Boolean formulas on FSM transitions and (2) uses a different state machine execution semantics than the one used in IEC 61499.

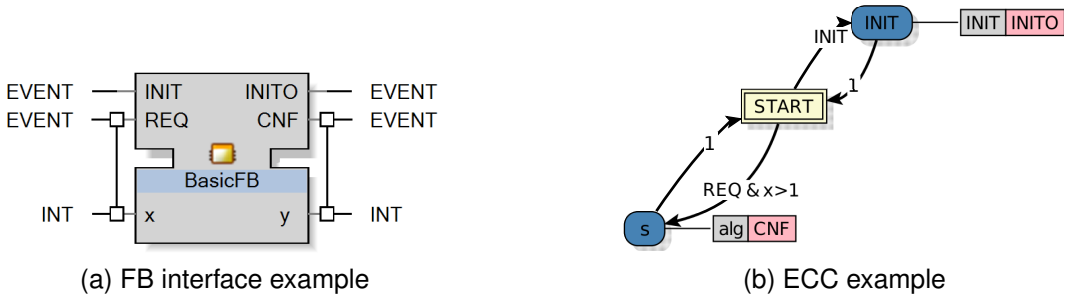
III. Problem Statement

A. IEC 61499 Standard

In the IEC 61499 standard applications are designed in the form of a network of interconnected FBs. Each FB has an *interface*, which defines possible input/output events and input/output variables. Variables can be, for example, Boolean, integer, or real, and can be associated with input and output events.

Basic FBs contain 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 variables) 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.

States can have a number of associated *algorithms* and output events. Algorithms are commonly implemented using



(a) FB interface example (b) ECC example

Fig. 1. Examples of an FB interface (left) and an ECC (right)

the *Structured Text* language and used, for example, for setting output variables values. An example of a FB interface is shown in Fig. 1a, an example of an ECC is shown in Fig. 1b.

B. Simplifications

In this work we will consider a simplified model of the ECC in which events are not taken into account and all variables are Boolean. Events can be in a straightforward way, dealing with non-Boolean variables is more complex.

An ECC in this paper is a six-tuple $\langle Y, X, Z, y_0, \phi, \delta \rangle$, where Y is a finite set of states, X is a set of Boolean input variables, Z is a set of Boolean output variables, $y_0 \in Y$ is the initial state, $\phi \colon Y \times 2^X \to Y$ is the transitions relation and $\delta \colon Y \to \{0,1\}^{|Z|}$ is the outputs relation. The outputs relation defines that each state is associated with a so-called algorithm, which in our simplified case is a function over output variables that transforms a Boolean string to another Boolean string. It is assumed that the initial state y_0 is always state 0.

An *execution scenario* s is a series of execution scenario elements s_i, where each element consists of a set of input variable values in_i and a set of output variable values out_i. For example, if there are three input and two output variables, $\langle 000, 00 \rangle, \langle 001, 01 \rangle, \langle 101, 11 \rangle$ is an execution scenario.

This paper is devoted to designing 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 input FB is first refactored in order to allow execution scenarios recording. A human designer supplies test cases. The FB application with the refactored FB is run on test cases, execution scenarios are recorded. Finally, the solution is generated using the proposed ECC inference algorithm.

The model inference algorithm we use is based on the pMuACO algorithm [6]. 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 considered optimization problem. The solution considered by the algorithm at any point in time is called a *candidate solution* or an *individual*.

A. Recording Execution Scenarios

In order to record execution scenarios, a refactoring of the FB system is performed. The input FB is replaced with a composite *ProxyLogger* FB; all input and output connections are copied. The *ProxyLogger* FB consists of an *InputLogger* FB, the input FB, and an *OutputLogger* FB. The *InputLogger* takes the input variable values, prints them, and passes the values unchanged to the input FB. The *OutputLogger* does the same with the output variable values of the input FB. Execution scenarios are recorded by running the FB application on a specially designed set of test cases.

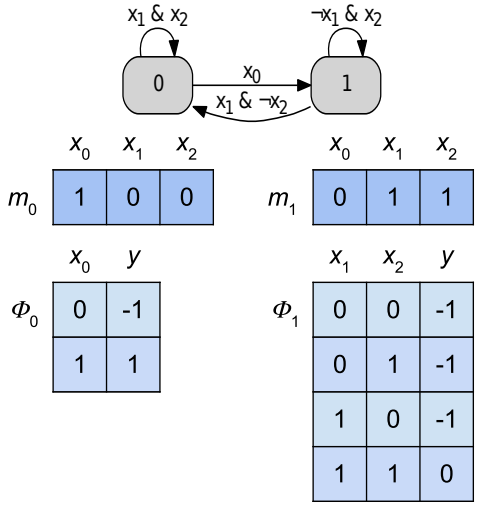


Fig. 2. An example of an ECC model and the representation of state 0. The "-1" entries mean that the corresponding transition is not present in the transition group.

B. Execution Control Chart Model

Our ECC model is represented as a set of states Y, where each state y_i has a set of transition groups T_i. Each transition group $t \in T_i$ has an associated Boolean array called the input variable significance mask m_t. If for some state $m_i = true$, then the input variable x_i is significant in this state. Each transition group also has a transition table Φ_t of $1 \times 2^{\Sigma m_t}$ elements, where Σm_t is the number of elements in m_t that are *true*.

Each j-th element of Φ_t stores the new state for that 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 $0101 = 2^5$. Examples of an ECC model and its representation are shown in Fig. 2: the ECC model is the state diagram on top and the representation of its state 0 is at the bottom.

The state algorithms are deduced from execution scenarios using a state labeling algorithm. Our state labeling algorithm is based on the same idea as the original state labeling proposed in [7] for learning Deterministic Finite Automata from labeled strings.

C. ECC Inference Algorithm

For ECC model inference we use the pMuACO algorithm [6]. The algorithm starts with a randomly generated initial solution and explores the search space using *mutation operators*, which usually 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* (FF). Before computing the FF value of a candidate solution, the algorithms for each state are determined using state labeling.

D. Mutation Operators

The following four mutation operators were used.

Change transition end state. For a randomly selected transition, the state y it leads to is changed to another state, which is selected uniformly at random from $Y \setminus \{y\}$.

Add or delete transitions. Transition groups of each state are modified with a fixed probability. A transition group is selected uniformly at random from the set of all transition groups of a state. It is randomly decided whether to add or delete a transition. *Adding a transition*. A guard condition is selected from the set of all guard conditions for which a transition is not defined. A transition is added that is marked with the selected guard condition and leads to a state selected uniformly at random from the set of all states. *Deleting a transition*. A randomly selected transition is deleted from one of the transition groups of the selected state.

Change significance mask of a transition group. This mutation operator works with a randomly selected transition group and makes one variable insignificant while making another significant. First we select a transition group t uniformly at random from the set of all transition groups that have at least one transition. Then we select $i, j : m_i = true \wedge m_j = false$. The mutation operator swaps significance mask values at selected positions: $m_i \leftarrow false$, $m_j \leftarrow true$.

Change significant variables set of a transition group. This mutation operator changes the number of significant variables in a randomly selected transition group. It is randomly decided whether to add or delete a variable.

Adding a significant variable. First, a randomly selected insignificant variable is made significant: $m_i \leftarrow true$. The size of the transition table is doubled. For each old transition pointing to some state y, we add two transitions to the new transition table: with $x_i = false$ and with $x_i = true$. Both new transitions point to the same state y. *Deleting a significant variable*. First, the randomly 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 count the number of added transitions and, if it is divisible by two, we keep the transition with $x_i = false$, otherwise the one with $x_i = true$.

E. State Labeling

Before computing the FF value we have to determine the algorithms associated with each state. Since all output variables are Boolean, it is sufficient to have one algorithm per state. We represent algorithms as strings of length $|Z|$ over the alphabet {"0", "1", "x"}. Let a_i be the algorithm associated with state y_i. The j-th character of the algorithm is associated with the j-th output variable. Algorithms have the following semantics: if $a_i^j = 0$ or $a_i^j = 1$, set $z_j \leftarrow a_j^i$; if $a_i^j =$ "x", preserve current value of z_j.

Below the state labeling algorithm is described. For each state we store a list of pairs of strings P. The algorithm consecutively processes all execution scenarios. Before processing each scenario the ECC is in its initial state. Input variable sets of the scenario elements are fed to the ECC one by one.

Suppose we are processing the k-th scenario element ($k > 0$) and the current state is y. First, the ECC makes the transition induced by the set of input variables in_k, changing the current state y to a new value. Second, if the k-th output variable set out_k is different from out_{k-1}, then the pair $\langle \mathrm{out}_{k-1}, \mathrm{out}_k \rangle$ is added to list P_y.

After all scenarios have been processed, algorithms for all states are determined according to the following algorithm. Labels (algorithms) for each state y are determined separately. We iterate over all characters in the algorithm string, each such character is also determined separately. A a 2×2 integer table M is used, initially it is filled with zeroes. We scan the list of pairs P_y and count how many times a "0" was replaced by "1", "1" replaced by "0" and so on. The value in $M_{1,0}$ is the number of times a "1" was replaced by zero, $M_{0,1}$ is the number of times a "0" was replaced by "1", and $M_{0,0} + M_{1,1}$ is the number of times the value of the i-th output variable has not changed. Then we use the following decision relation:

$$C = \{\langle \text{"0"}, M_{1,0} \rangle; \langle \text{"1"}, M_{0,1} \rangle; \text{"x"}, M_{0,0} + M_{1,1} \rangle\}$$

The i-th character of the algorithm string is selected as the argument of the maximum value of C.

F. Fitness Function

The FF we used consists of three components:

$$F = 0.9F_{\text{ed}} + 0.1F_{\text{fe}} + 0.0001F_{\text{sc}}.$$

The F_{ed} component is based on string *edit distance* and measures the similarity of the output variables values' sequences that are recorded in the execution and scenarios and the ones acquired using the candidate solution. F_{fe} is based on the position of the *first error* the candidate ECC model makes, and F_{sc} is the number of *state changes*. The maximum possible value of the FF is 1.0001.

V. Experiments

Experiments were performed for one of the basic FBs of the Pick-and-Place (PnP) manipulator project [8]. The FBDK[2] development environment was used for recording execution scenarios and post-experiment simulation. A screenshot of the PnP system is shown in Fig. 3. The system has two horizontal

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

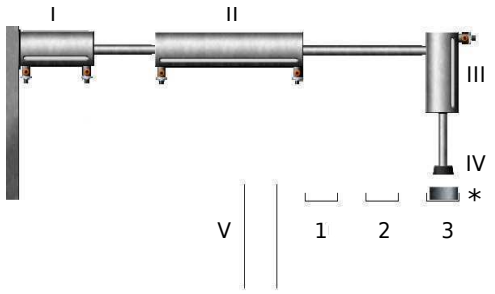


Fig. 3. Screenshot of one the Pick-and-Place manipulator system implementations

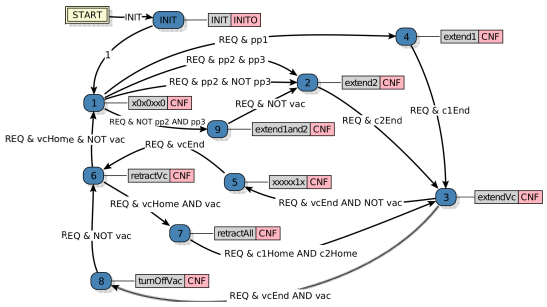


Fig. 4. One of inferred ECC models

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).

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. The FB has the following Boolean input variables: c1Home/c1End (cylinder I is in the leftmost/rightmost position), c2Home/c2End (cylinder II is in the leftmost/rightmost position), 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 (extend/retract horizontal cylinder II), vcExtend (extend vertical cylinder III), vacuum_on/vacuum_off (turn suction unit on/off).

For comparison we confront the generated ECC with the manually created one, further referred to as *original*. We shall note, however, that the original ECC is not anyhow used in the reconstruction process.

A. Inferring ECC Models

The experiment plan was as follows: (1) record execution scenarios, (2) use the proposed approach to infer a ECC model compliant with all scenarios, (3) test all inferred ECC models in simulation using FBDK. We recorded ten execution scenarios, 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' is a scenario where first the manipulator is given piece number one, then piece number two, and, finally, piece number three. The ten test cases are as follows: '1', '1-2', '1-2-3', '2', '2-1', '2-3', '3', '3-2', '3-2-1', and also '123' (all work pieces are placed simultaneously). We used a machine with a 64-core AMD Opteron(TM) 6378 @ 2.4 Ghz. processor the pMuACO algorithm was run on 16 cores.

The experiment was repeated 20 times. It took our algorithm an average of about 4.5 hours to find a perfectly fit solution. Afterwards, all constructed solutions were tested in simulation using FBDK. The ECC model is automatically converted to a IEC 61499 FB format file. In simulation testing we verified that all constructed solutions can correctly handle all test cases. One of the constructed ECC models is shown in Fig. 4. Algorithms that are identical to the ones used in the original ECC are called by their names.

VI. Conclusion

We have presented an approach for reconstructing ECCs of basic IEC 61499 FBs with Boolean input/output variables. The feasibility of this approach was demonstrated on one basic FB designed for performing centralized control of the Pick-and-Place manipulator system.

For future work we plan do add support for integer and real input/output variables, and modify the proposed approach for inferring simple FBs from expert data. Such data can be acquired by implementing a user interface so that the expert can control the plant manually.

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.

References

[1] 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, March 2014.

[2] 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, Apr. 2008.

[3] 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 International Conference on Industrial Informatics, 2009. INDIN 2009.*, June 2009, pp. 715–720.

[4] 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.

[5] I. P. Buzhinsky, V. I. Ulyantsev, D. S. Chivilikhin, and A. A. Shalyto, "Inducing finite state machines from training samples using ant colony optimization," *Journal of Computer and Systems Sciences International*, pp. 256–266, 2014.

[6] D. Chivilikhin and V. Ulyantsev, "Extended finite-state machine inference with parallel ant colony based algorithms," in *Proceedings of the International Student Workshop on Bioinspired Optimization Methods and their Applications (BIOMA'14)*, 2014, pp. 117–126.

[7] 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, Jul. 2005.

[8] 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'12)*, Sept 2012, pp. 1–7.