Testing of Control Programs in Distant Education [⋆]

Richard Šusta [*] **Pavel Burget** [**]

[*] *DCE-Prague: Department of Control Engineering, Faculty of Electrical Engineering, Prague, Czech Republic, (Tel: +420 22435-7359; e-mail: richard@susta.cz)*
[**] *DCE-Prague, (Tel: +420 22435-7610; e-mail: burgetpa@control.felk.cvut.cz)*

Abstract: The paper deals with automatic testing of programmable logical controller (PLC) programs in a distant education. Students control a physical model, which that has also its virtual counterpart. While one user is connected to the physical model, others debug programs with the aid of the virtual model. We discuss the structure of models, the organization of education, and the testing process of student's programs. Finally, we present a new theory of δ-graphs used for conversion programs into a timed abstraction of PLC suitable for testing, as the main contribution of this paper.

Keywords: Educational aids; Verification; Automatic testing; Programmable logic controllers; PLCs; Timed automata.

1. INTRODUCTION

Laboratory experience is considered to be a critical component of all technical coursework, which is especially valid for a distant education in control engineering. Physical models of simplified technological processes represent time-tested teaching aids. Undoubtedly, the students will better understand theoretical parts, if they can practically apply their knowledge by creating their own control programs. However, there are many difficulties associated with providing of distant access to physical models without direct supervision of teachers.

In the first place, selected physical models for distant education must be fully "student-proof", i.e. practically indestructible by unskillful beginners — some students perform experiments that go against of common sense. Additionally, the models should constitute meaningful systems, at least distantly relative to real technologies. Finally, we need only such processes, initial states of which are always easily controllable, i.e., if a model begins in a certain initial state, it will again return to that state after undergoing some predefined control actions.

We can easily meet all conditions above by virtual models, but physical models undoubtedly offer more gains in terms of touches with reality and stronger motivation for students.

Figure 1 depicts one of our physical models suitable for distant education, which was described in detail in [Burget et al.(2004.)]. Colored ping-pong balls are released one by one from the upper storage bin. They are picked by two way valve that alternately pushes out two horizontal rods. When the lower rod is pushed out, the optical sensor

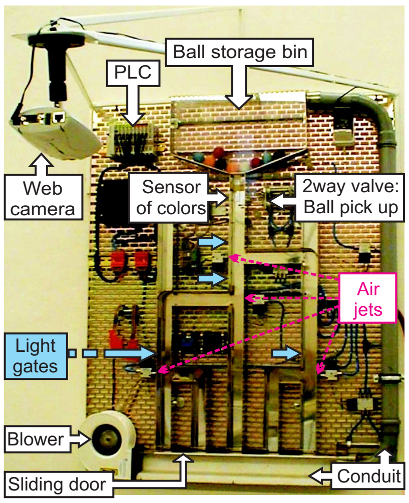

Fig. 1. Physical model of sorting of colored balls

recognizes the color of a ball that has just felt down from the bin. If the rods are switched again, i.e. the upper rod is pushed out, the ball is released. It rolls down through tree-like chutes, in which it is directed by air blows from the air jets placed at forks. Its positions are indicated by four light gates. The chutes are ended by the sliding doors. When the doors are shifted, balls fall down into the conduit. The blower moves them back into the upper storage bin. Students should sort balls according to colors. They write programs in a free demo version of WAGO-IO-PRO development environment that allows creating simple ladder diagrams suitable for training purposes.

Programs can be debugged by several ways, depicted in Figure 2. D level means a direct connection to the model from laboratory. A remote connection to the model with the aid of the web server, marked as R level, represents

[⋆] This work is supported by euSophos 2C06010 project granted by the Ministry of Education.

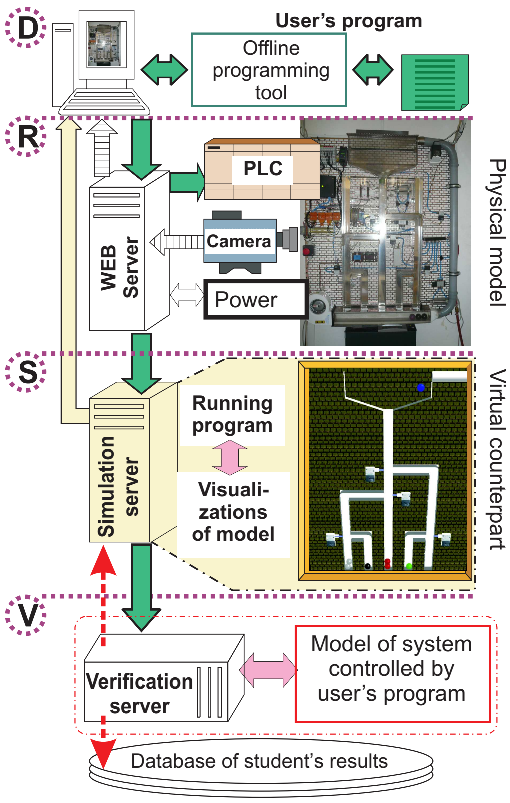

Fig. 2. Flow of user's programs

a more flexible approach. Students upload program files to the web server, which downloads them into the programmable logical controller (PLC) connected to the physical model, and they watch the behavior of the physical model by a real-time web camera.

We utilize this solution successfully in our courses since 2002 year. Nowadays, we have 3 different models fully suitable for remote connections. The sorting model described above is one of them. The remote approach on R level has many advantages, but also a drawback — one cycle of a model operation (i.e. rolling down of all balls through chutes) always requires exclusive access. Therefore, multiple students cannot efficiently share the model at a time.

We improved the situation by creating the simulation server, see S level in Figure 2. Meanwhile one user utilizes the physical model other students can debug their programs with the aid of its virtual counterpart, which has nearly the same behavior as the corresponding physical model. There are certainly some minor differences for special cases of ball positions, but the virtual model emulates the physical model exactly for all correct control programs.

After an initial login process, students download their web client modules, if they do not have them yet. The modules are sent directly to their computers from the simulation server. The client module contains mainly visualization support and graphical data of all static elements. If a student uploads his/her control program to the simulation server, he/she will receive only responses with data describing both the movements of balls and the states of main program variables. Therefore, huge data blocks are downloaded only in the beginning, all following network communication is minimized.

The simulation server offers even more functionality than the physical model. Students can replay received data streams several times. If they interrupt visualizations in an interesting state, it is possible to step forward or backward in time and to watch corresponding PLC inputs and outputs.

The number of users of the model was increased but it has also prolonged the validating phase. After students submit their final programs to obtain credits, teachers must again recheck their works with the aid of the virtual or physical model, which entails long exhausting operations. Furthermore, it sometimes happens under unfavorable conditions that erroneous programs successfully undergo teacher's brief check. The simulation server certainly finds out automatically if uploaded programs have terminated with correct results, but single simulations do not naturally reveal all errors.

We decided to add the verification server, see V level in Figure 2, which tests submitted control programs. It reveals more errors, therefore, its results can be also considered as more reliable information for assigning credits to students. The testing process needs certainly a faster modeling method than the calculation of exact balls movements performed by the simulation server for graphical visualizations.

1.1 Outline of Following Sections

As the main contribution of this paper, we describe the testing method for V level in Figure 2, based on timed abstractions. In the next section, we first define *timed abstraction of PLC program*, abbreviated as TAPLC, which represents the main ground for simulations and tests. In Section 3, we introduce TAPLCTrans method that transforms PLC source code into TAPLC form. Then in Section 4, we describe our testing method based on TAPLC that is used on V level in Figure 2. Finally, we look at our future plans in the conclusion.

2. TIMED ABSTRACTION OF PLC PROGRAM

In this section, we first resume several properties of utilized PLC controller that are important for the following paragraphs. Finally, we define TAPLC.

PLC controller used in our model operates in classical cyclic manner, which assures synchronizing I/O data with the evaluations of its program. One *PLC scan* consists of consequent performance of three following steps, see Figure 3:

Input scan: Hardware inputs X are polled or sampled respectively and their values are stored in inner PLC memory called *Input image*;

Program scan: $\mathcal{C}$ program code is executed once. It calculates new outputs and writes them into inner PLC memory called *Output image*; and

Output scan: As soon as $\mathcal{C}$ program is terminated, the values in output image are copied into corresponding peripherals U.

PLC scans are generally irregular, because a new PLC scan begins after finishing the previous one and the length of the scan depends on evaluation time of its instructions.

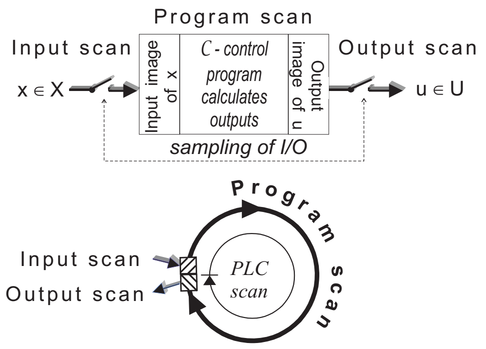

Fig. 3. PLC scan cycle

But PLC programs behave as event driven systems, thus, we may suppose without loss of general character that PLC scans are performed at regular time intervals τ, where τ value should be much less than a permissible maximal PLC time response to input signals.

PLC programs frequently use software timers updated only during program scans. If we suppose regular PLC scan, then a software timer with t_p preset time will finish its timing only after $t = n * \tau$ time, where n is an integer number satisfying $n * \tau \geq t_p > (n-1) * \tau$. Therefore, it is possible to replace software timers by appropriate scan counters.

In following definitions, variables will be emphasized by membership in $\mathcal{V}$, the set of all variables.

Definition 1. A scan counter $\mathbf{s}(f, g, \mathrm{T})$ is an automaton of Moore's family with one (enabling flag) input $f \in \mathcal{V}$, one output gate $g \in \mathcal{V}$, sets of states $Q = \{q_0, q_1, ...q_T\}$, $|Q| = T + 1$, where $T > 0$ is a given integer constant. Q has has always q_0 as its initial state. Output g equals to 1 only in q_n state, otherwise $g = 0$, and δ transition function is defined by:

$$\delta(q_i, g) = \begin{cases} q_{i+1} & : \text{ if } f \neq 0 \text{ and } i < n \\ q_n & : \text{ if } f \neq 0 \text{ and } i = n \\ q_0 & : \text{otherwise} \end{cases}$$

A scan counter can be easily implemented by incrementing an accumulator according to f flag value.

Definition 2. Let S be a finite set of m scan counters, $S = \{\mathbf{s}_1(f_1, g_1, \mathrm{T}_1), \mathbf{s}_2(f_2, g_2, \mathrm{T}_2), ...\mathbf{s}_m(f_m, g_m, \mathrm{T}_m)\}$. We define flag($S$) and gate($S$) sets by

$$\text{flag}(S) \overset{df}{=} \{f_i \in \mathbf{s}_i \mid \mathbf{s}_i \in S,\ i = 1..|S|\}$$
$$\text{gate}(S) \overset{df}{=} \{g_i \in \mathbf{s}_i \mid \mathbf{s}_i \in S,\ i = 1..|S|\}$$

We must also distinguish two initializations:

(1) All variables are always initialized to specified values when a new $\mathcal{C}$ program is downloaded into PLC;
(2) For safety reasons PLC operating systems perform an additional partial initialization called *pre-scan* before beginning the first program scan of $\mathcal{C}$. Some variables belonging to outputs of selected *non-retentive* instructions are rewritten by their default values, usually by zeros. The set of pre-scan initializations results from $\mathcal{C}$ structure.

Software timers can also be retentive or non-retentive. PLC instruction sets offer several timer types, e.g. pulse timers, timers on delay, or timers off delay.

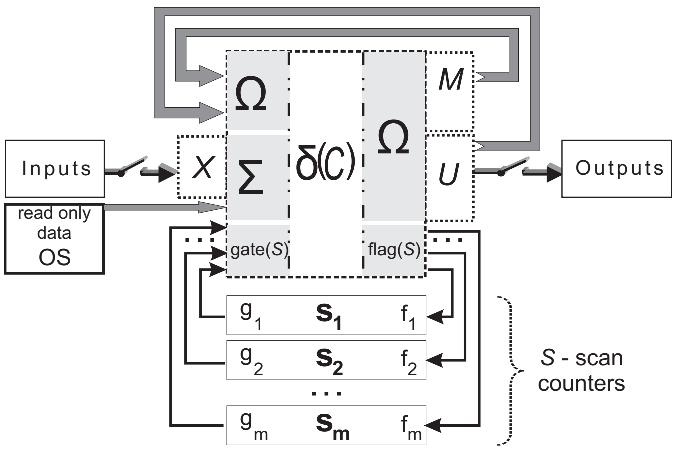

Fig. 4. Timed abstraction of PLC

Hypothesis 3. Let $\mathcal{C}$ be a PLC program with a regular program scan. If all its non-retentive software timers have constant preset times, then $\mathcal{C}$ can be modified to functionally equivalent $\mathcal{C}'$, in which all non-retentive timers are replaced by scan counters.

The simple proof is performed by the constructions of such replacements, but a proposition valid for all possible PLC programs needs more detailed assumptions about hypothetical timer instructions exceeding the limited size of this paper. Thus, we have presented only the hypothesis.

For correctness of the following mathematical notations, we need $\alpha(.)$ construction.

Definition 4. Let $V \subseteq \mathcal{V}$ be a finite ordered set of variables, then $\alpha(V)$ creates an alphabet set containing n-tuples of all possible combinations of the values that can be assigned to variables in V, where $n = |V|$. If $|V| = 0$, then $\alpha(V)$ is defined as an empty set.

Now we can finally present the definition of a timed abstraction of PLC program.

Definition 5. (TAPLC). Given a tuple

$$\mathbf{A} \overset{df}{=} \langle \tau, \Sigma, \Omega, S, \Omega(0), \delta(\mathcal{C}) \rangle$$

where $\tau > 0$ constant represents regular PLC scan time, $\Sigma \subseteq \mathcal{V}$ and $\Omega \subseteq \mathcal{V}$ are finite ordered disjunctive sets of input and output variables, $\Sigma \cap \Omega = \emptyset$, S is a finite ordered set of scan timers, $\Omega(0)$ describes all initial values of Ω variables and their pre-scan initializations, if any. If $\delta(\mathcal{C})$ mapping satisfies

$$\delta(\mathcal{C}) : \alpha(\Sigma) \times \alpha(\Omega) \times \alpha(\text{gate}(S)) \rightarrow \alpha(\Omega) \times \alpha(\text{flag}(S))$$

then $\mathbf{A}$ is called an *timed abstraction of a PLC program* (abbreviated as TAPLC).

In other words, TAPLC represents the results of program scans performed in regular time intervals τ with the aid of $\delta(\mathcal{C})$ mapping and scan counters S.

If we have a given PLC, then Ω of corresponding TAPLC consists of U PLC outputs (see Figures 4 and 3) and M internal PLC memory, i.e. $\Omega = U \cup M$.

Σ set of TAPLC consists mainly of X inputs of PLC sometimes extended by read-only data from PLC operating system, if they are applied. But their set is frequently empty for the presented model, thus, $\Sigma = X$ in many cases. All timers instructions with constant preset times are converted into scan timers in S set.

In each program scan, we first update all enabled scan counters, then we evaluate $\delta(\mathcal{C})$. But if there were no changes of Σ, Ω, and gate(S) values, we can skip $\delta(\mathcal{C})$ evaluation and use last values of Ω, see algorithm on Page 6. In this way, TAPLC accelerates simulations of PLCs.

The second advantage of TAPLC concerns its close relation to a classical timed automaton [Alur and Dill(1994)]. Testing of such timed automaton is mostly performed by its conversion into other models. TAPLC utilizes properties of PLCs to bypass the first step and to create directly easily testable automaton, which moreover expresses better PLC behavior.

TAPLC definition does not contain any construction algorithm; it will be the main topic of the next section.

3. TAPLCTRANS ALGORITHM

In this section, we introduce TAPLCTrans algorithm converting a subset of $\mathcal{C}$ programs into special oriented graphs.

Definition 6. (δ-node). An *δ-node* is a tuple $\mathbf{d} = \langle I, O, \delta \rangle$, where $I \in \mathcal{V}$ and $O \in \mathcal{V}$ are finite disjunctive ordered sets of $\mathbf{d}$ inputs and outputs, $I \cap O = \emptyset$, and δ represents a mapping $\delta : \alpha(I) \rightarrow \alpha(Q)$.
If $|I| = 0$ and $|O| = 1$ then $\mathbf{d}$ is called a *beginning δ-node.*
If $|I| = 1$ and $|O| = 0$ then $\mathbf{d}$ is called an *end δ-node*, otherwise $\mathbf{d}$ is called a *middle δ-node.*
If P is a given finite set of δ-nodes, then the sets of all inputs and outputs are defined by:

$$\mathrm{in}(P) \overset{df}{=} \bigcup_{i=1}^{|P|} I_i, \quad \mathrm{out}(P) \overset{df}{=} \bigcup_{i=1}^{|P|} O_i$$

where $I_i, O_i \in \mathbf{d}_i \, \langle I_i, O_i, \delta_i \rangle \in P$.

TAPLCTrans functionality does not depend on types of variables. Therefore, it assumes typeless variables for simplicity and lets the distinguishing of numeric types to operations in δ-nodes. For verification processes, numeric ranges of variables can be easily reconstructed latter by analysis of δ-nodes.

Definition 7. Let $\mathbf{A} = \langle \tau, \Sigma, \Omega, S, \Omega(0), \delta(\mathcal{C}) \rangle$ be a given TAPLC. Ordered set $Z = \Sigma \cup \Omega \cup \mathrm{flag}(S) \cup \mathrm{gate}(S)$ is called a *variable set of TAPLC.*

Definition 8. Let Z be a given variable set of TAPLC and $Y \subseteq Z$ any its ordered subset. We will write beg(Y) and end(Y) for such ordered sets of beginning and end δ-nodes that satisfy $Y = \mathrm{out}(\mathrm{beg}(Y))$ and $Y = \mathrm{in}(\mathrm{end}(Y))$. We will call beg$(Y)$ and end(Y) *beginning and end δ-node sets.*

Notice that end δ-nodes also exist for inputs $\Sigma \subseteq Z$ of TAPLC, because they are stored in PLC input image memory, thus, programmers may rewrite any input as other variables; it is sometimes used for its readdressing.

Definition 9. (δ-graph). Let Z be a given variable set. δ-graph is an acyclic oriented graph defined by

$$\mathcal{G} = \langle \Xi, N, \Theta, E, \epsilon \rangle$$

tuple, where N is a finite set of middle δ-nodes, $\Xi = \mathrm{beg}(Z)$ and $\Theta = \mathrm{end}(Z)$ are ordered sets of beginning and end δ-nodes, E is a finite set of oriented edges, and ϵ is mapping that assigns to each edge $e \in E$ exactly one ordered pair, which consists of an output and an input,

$$\epsilon : E \rightarrow \mathrm{out}(\Xi \cup N) \times \mathrm{in}(\Theta \cup N)$$

such that any input in$(\Theta \cup N)$ is used at most in one pair. If ϵ mapping assigns exactly one edge to each end δ-node $d_i \in \Theta$, $i = 1..|\Theta|$, then $\mathcal{G}$ is called *complete δ-graph in Z*, otherwise $\mathcal{G}$ is called an *incomplete δ-graph in Z.*

Remarks to definition 9:

(1) *Variable set:* Notice that $|\Xi| = |\Theta| = |Z|$ for any δ-graph. Moreover, a complete δ-graph always assumes assigning of all variables in Z, because it is a necessary condition for compositions. If we create an incomplete δ-graph for some instructions that has assigned only a subset of Z, then we easily complete such δ-graph by adding *primitive edges* that will connect all unused in(Θ) inputs with corresponding out(Ξ) outputs.
(2) *Implementation:* δ-graphs can be effectively implemented as macro blocks of symbolic equations connected by edge links, thus, e.g. a long logical operation is represented either by one δ-node or by several δ-nodes, as required. Primitive edges and unused beginning/end δ-nodes are not stored. They are added and also removed automatically, if they are required by a manipulation algorithm.

Definition 10. Let $\mathcal{G}_1 = \langle \Xi_1, N_1, \Theta_1, E_1, \epsilon_1 \rangle$ and $\mathcal{G}_2 = \langle \Xi_2, N_2, \Theta_2, E_2, \epsilon_2 \rangle$ be two δ-graphs in Z. We define their composition $\mathcal{G} = \mathcal{G}_1 \circ \mathcal{G}_2$ by the following:

$$\mathcal{G} = \langle \Xi_1, N_1 \cup N_2 \cup N_{12}, \Theta_2, E_1 \cup E_2, \epsilon_1 \cup \epsilon_2 \rangle$$

where N_{12} set of middle δ-nodes satisfies $|N_{12}| = |Z|$ and also $|Z| = |\Theta_i| = |\Xi_i|$, $i = 1, 2$, and it is defined as:

$$N_{12} = \left\{ \begin{array}{c} \mathbf{d}_i \, \langle \mathrm{in}(\theta_i), \mathrm{out}(\xi_i), \epsilon : \mathrm{in}(\theta_i) \rightarrow \mathrm{out}(\xi_i) \rangle \\ \text{such that } \theta_i \in \Theta_1, \xi_i \in \Xi_2, i = 1..|Z| \end{array} \right\}$$

In other words, the composition connects two δ-graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by joining Θ_1 end δ-nodes in $\mathcal{G}_1$ with corresponding Ξ_2 beginning δ-nodes in $\mathcal{G}_2$. The joined δ-nodes have one input and one output and they contain δ mappings corresponding to no-operation instructions, i.e. their single inputs are assigned to their single outputs. Such no-operation nodes can be certainly further removed by a minimizing algorithm.

Proposition 11. The composition of δ-graphs in Z is an associative operation.

Proposition 12. The composition of two complete δ-graphs in Z is also complete δ-graph.

If δ-graphs are defined in the same Z, then we connect end δ-nodes with corresponding beginning δ-nodes, therefore, the order of $\circ$ compositions is irrelevant. The completeness of result is given by $\circ$ members according to Definition 9. Propositions 11 and 12 does not hold for δ-graphs in different Z_1 and Z_2 variable sets.

Proposition 13. Let r_1 and r_2 be instructions expressible by $\mathcal{G}_1$ and $\mathcal{G}_2$ complete δ-graphs in Z. If r_{12} represents composed instruction given by executing first r_1 and then r_2, then $\mathcal{G} = \mathcal{G}_1 \circ \mathcal{G}_2$ is complete δ-graph of r_{12}.

Notice the important assumption of the existence of complete δ-graphs in Z for r_1 and r_2. *Proposition 13 does* *not hold for incomplete δ-graphs*, in which some data assignments can be absorbed. The proof also results from the same assumption. If instructions have complete $\mathcal{G}_1$ and $\mathcal{G}_2$ δ-graphs, then they correspond to $\alpha(Z) \rightarrow \alpha(Z)$ mappings, thus, their composition will be also a mapping: $\alpha(Z) \rightarrow \alpha(Z)$.

Proposition 14. Let $\mathcal{C}$ be a program running in a regular PLC scan. If complete δ-graphs in a given variable set Z exist for all instructions executed during the scans, then $\mathcal{C}$ can be converted into complete δ-graph in linear time in the number of $\mathcal{C}$ instructions.

Propositions 14, 13, and 11 allow an effective conversion in linear time in size of its source code, even if a program has exponential execution time due to calling many subroutines. We use associativity of the composition. First, we convert subroutines, then we just insert them. The approach is the same as for a binary PLC and it was described in [Šusta(2003)], pages from 73 to 96, where are also conversions of PLC instructions to transfer sets, predecessors of δ-graphs.

But Propositions 14 contains the assumption that complete δ-graphs exist for all instructions in $\mathcal{C}$. It rises an important question, which operations have no direct δ-graph representations. Generally speaking, δ-graphs contain $\alpha(Z) \rightarrow \alpha(Z)$ mappings, thus, they do not exist for operations with unpredictable results, e.g. instructions with side-effects involving variables outside Z. Moreover δ-graphs hardly represent operations having too many possible inputs and outputs, e.g. indexes or pointers.

The instructions that perform backward jumps in programs constitute a specific problem, because they are not directly convertible into δ-graphs. Program loops can be only represented by unexpandable δ-nodes that include all their operations. Fortunately, loops are considered as a bad programming style in PLC programs for many reasons, e.g. worse trouble shooting and longer PLC scan. In our educational challenge, students are asked to replace complex loops by other constructions.

Proposition 15. A given $\mathcal{C}$ program running in regular PLC scan with τ period is convertible into a TAPLC form if and only if $\mathcal{C}$ has a δ-graph representation.

The proof is performed on the basis of constructions and this will be briefly outlined. From a given δ-graph, we can surely construct a TAPLC. Scan counters are moved to outside blocks, the remaining part of δ-graph becomes $\delta(\mathcal{C})$ of TAPLC. On the contrary, if we have a TAPLC, then we can express its $\delta(\mathcal{C})$ in all cases by $\mathcal{G}_p$ δ-graph that contains one δ-node with $\delta(\mathcal{C})$ mapping. Scan counters are evaluated before $\delta(\mathcal{C})$, thus we create their $\mathcal{G}_s$ δ-graph. Finally, we compose $\mathcal{G}_s \circ \mathcal{G}_p$ into one δ-graph.

Remark: Many manipulation algorithms exist for δ-graphs. For instance, it is sometimes possible to remove spare operations, to join several nodes into one, or on the contrary to expand a δ-node. The algorithms are omitted to reduce the size of this paper.

3.1 Related works to δ-graphs

Presented δ-graphs are the improved version of elder transfer sets defined only for PLC programs with binary data, see [Šusta(2003)]. Even if transfer sets can be extended to other data types, see [Šusta(2004)], the main disadvantage of transfer sets concerns single outputs of their operations, which complicate conversions of more complex programs. The idea of δ-graphs combines classic PLC function block diagrams, defined by IEC 1131–3 standard, with compositional techniques for program analysis that were described for instance in [Nielson et al.(1999)].

δ-graphs were developed mainly for programs running in PLC scan cycles rarely containing program loops and jumps. Such assumption nearly exclude an application of δ-graphs in other programming environments. It will be also a reason why we have not found any publications directly related to δ-graphs, even after having surveyed many papers in this area. Thus, δ-graphs might be our original contribution.

4. TESTING OF TAPLC PROGRAM

Unlike the situation in industry, teaching models represent an ideal situation for the application of verification methods. The models have usually known simple structure and small number of inputs and outputs. Moreover, methods laboriously developed by us will be further used for a large number of beginner's programs frequently containing "textbook" errors. Besides, an automatic testing is nearly unavoidable here, especially in a distant education, otherwise qualified teachers would waste their time by watching control actions of thousands user's programs.

On the contrary, beginner's programs have drawbacks that often include many experimental parts, e.g. undeleted previous versions or auxiliary blocks with operations intended only for debugging purposes.

The simulation represents our primary testing tool. User's programs are always converted into TAPLC and connected to the virtual model of sorting balls. After the programs have successfully sorted all ten balls, their in-depth tests become meaningful — student's programs have now expected functionality and we can make assumptions about their internal structure.

Before any verification attempt we must first remove unused or debugging parts that have no influence on outputs to simplify tests. Fortunately, δ-graphs allow deleting such irrelevant blocks.

Let be given a $\mathcal{G}$ δ-graph. In accordance with the theory of oriented graphs, we will call a *path in* $\mathcal{G}$ any sequence of nodes and edges, $p = n_0 e_1 n_1 ... e_m n_m$, $m \geq 1$, such that $n_i, e_i \in \mathcal{G}$ and e_i edge connects an output of n_{i-1} node with an input of n_i node. We will call n_0 and n_m as *beginning and end nodes* of p path.

Definition 16. Let be given $\mathcal{G} = \langle \Xi, N, \Theta, E, \epsilon \rangle$ a δ-graph in Z variable set. The concatenation $p_a p_b$ is called a *scan path in* $\mathcal{G}$, if p_a and p_b are two paths or scan paths in $\mathcal{G}$ such that $\theta_i \in \Theta$ is a end node of p_a and $\xi_i \in \Xi$ is a beginning node of p_b, and $\text{in}(\theta_i) = \text{out}(\xi_i) = v \in Z$.

In other words, a scan path allows an exception in regular alternating of an edge and a node in a path. In a scan path, an end δ-node of $\mathcal{G}$ can be immediately followed by the beginning δ-node of $\mathcal{G}$ that belongs to the same variable

$v \in Z$. By this way, scan paths describe how a value of the variable can influence the following programs scans.

Using scan path, we define an irrelevant δ-node. The definition also suggests construction algorithm for a set of all such nodes.

Definition 17. Let be given $\mathcal{G} = \langle \Xi, N, \Theta, E, \epsilon \rangle$ a δ-graph in Z variable set. We say that a middle δ-node $\mathbf{d} \in N$ is irrelevant with respect to some given subset $H \subseteq \Theta$ of end nodes, if no scan path of $\mathcal{G}$ beginning in $\mathbf{d}$ contains any $h \in H$.

Algorithm for irrelevant nodes:

(1) *Initialization:* We mark all end δ-nodes specified by a given H subset.
(2) We mark all unmarked δ-nodes, outputs of which are connected to inputs of already marked δ-nodes. If a beginning δ-node in Ξ is marked, we also mark its corresponding end δ-node in Θ.
(3) We repeat the previous step, until no δ-node is possible to mark.
(4) All unmarked δ-nodes are irrelevant with respect to H and can be deleted with their edges, because they have no influence on H values.

Notice that the algorithm does not follow oriented edges that begin in outputs of marked δ-nodes. It checks only δ-nodes connected to inputs of already marked nodes to find out all operations that have affected them. $\mathcal{G}$ δ-graph contains a finite number of δ-nodes, therefore, the algorithm is sure to terminate after a finite number of steps. If $\mathcal{G}$ edges are implemented as bidirectional links, the computational complexity of the algorithm will be $O(n)$; it depends linearly on the number of edges in $\mathcal{C}$. If we select H that contains δ-nodes of controlled PLC outputs U, then a program is reduced by deleting auxiliary parts.

To verify a program, we should test its responses to all possible sequences of 10 balls with 5 different colors: $10!/(2!)^5 = 113400$. But we do not need exact simulations of ball movements. If a ball is released, it crosses light gates in times given by probability distributions which can be approximated by ranges of minimum and maximum delays.

For the purposes of tests we replace the simulation of our physical model by the second TAPLC $\mathbf{A}_M$ having time constants of its scan counters S^r_M specified by the ranges of minimum and maximum values. We omit its simple definition to reduce this paper.

Let us write $\mathbf{A}_P$ for TAPLC of the tested program. Then, the connection of program and model TAPLCs is described by two equations:

$$\begin{aligned} \mathbf{A}_P &= \langle \tau, \Sigma_P = X, \Omega_P = U \cup M_P, S_P, \Omega_P(0), \delta(\mathcal{C}_P) \rangle \\ \mathbf{A}_M &= \langle \tau, \Sigma_M = U, \Omega_M = X \cup M_M, S^r_M, \Omega_M(0), \delta(\mathcal{C}_M) \rangle \end{aligned}$$

The testing algorithm can be roughly described as a classical search of an automaton state space, which was improved by advancing simulation time, if both TAPLC are only waiting for scan counters.

Outline of our testing algorithm:

(1) *Initialization:* Initialize $\mathbf{A}_P$ and $\mathbf{A}_M$ and store the values of their variable sets (see Definition 7).
(2) *Test loop:* Update scan counters S_P.
(3) If any input of $\mathbf{A}_P$ has changed, evaluate $\delta(\mathcal{C}_P)$.
(4) Update range scan counters S^r_M.
(5) If any input of $\mathbf{A}_M$ has changed, evaluate $\delta(\mathcal{C}_M)$.
(6) *Check sorting:* Terminate with a sort error, if $\mathbf{A}_M$ announces a failure in sorting.
(7) *Check waiting for scan counters:* If there are any changes in the values of $\mathbf{A}_P$ and $\mathbf{A}_M$ variable sets, then store new values and go to step 2.
(8) Find out an enabled range/scan counter with minimum remaining time. Fork the process to more test paths, if a scan counter finishes between range values of some range scan counter. Terminate with a deadlock error, if no counter is enabled.
(9) *Advancing simulation time:* Let k be a remaining count of the selected counter. Add $k - 1$ to all range/scan counters and then go to step 2.

After releasing a ball, our test algorithm also stores $\mathbf{A}_P$ states in an associative memory. If a state was already stored here, we can skip rechecking of previously tested path. If an error was found, it is approximated by the graphical simulation with the aid of the virtual model and data stored during testing.

The testing algorithm selects ball combinations in a random order and it terminates after a predefined time, thus, it does not always check all and any possible input sequences. Even if a partial test does not reveal all errors, it finds out more mistakes than any brief teacher's checks. Its improvement is a matter for our future research.

5. CONCLUSION AND FUTURE PLANS

At time of writing this paper our physical model was reconstructed to a robuster version with rod chutes that are not sensitive to dust as plate chutes. The new model sorts 14 balls with 7 different colors. Its virtual counterpart is much easier, because the chutes are now narrower and the balls rolls down on nearly deterministic paths. Testing of user's control programs will take more time, but all methods described in this paper are also valid for the improved model. *The virtual model is now available on web page* http://dce.felk.cvut.cz/virtualmodel/ .

REFERENCES

[Alur and Dill(1994)] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Burget et al.(2004.)] P. Burget, O Dolejš, Z. Hanzálek, and B. Kirchmann. Remote programming of control systems. In *2nd IFAC Workshop on Internet Based Control Education 2004 [CD-ROM],*. Grenoble: Laboratoire d'Automatique de Grenoble, 2004.

[Nielson et al.(1999)] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999. ISBN 3–540–65410–0.

[Šusta(2004)] Richard Šusta. Low cost simulation of PLC programs. In *7th IFAC Symposium on Cost Oriented Automation COA 2004, Gatineau (Québec) Canada*, pages 219–224. Université du Québec en Outaouais, 2004.

[Šusta(2003)] Richard Šusta. *Verification of PLC Programs.* PhD thesis, CTU-FEE Prague, May 2003. avail. at http://dce.felk.cvut.cz/susta/.