LOW COST SIMULATION OF PLC PROGRAMS

Richard Šusta *

* *Department of Control Engineering, Faculty of Electrical Engineering, Prague*
http://dce.felk.cvut.cz/susta/,
susta@control.felk.cvut.cz

Abstract: The paper discusses the conversion of PLC (programmable logical controller) programs into forms suitable for their emulation by any software tool that includes some programming language capable of evaluating mathematical formulas, if-then instructions, and time tests, in the case of timers. The method offers advantage of low cost portability into a whole range of environments because it does not require the embodiment of large additional programming support for the simulation of various PLC instructions. The conversion, either of whole PLC program or its part, can also be utilized for auxiliary tests when a finished PLC program is transformed into a new hardware. *Copyright © IFAC 2004*

Keywords: PLC, emulation, conversion, transfer sets

1. INTRODUCTION

Some PLC emulators are offered by their manufactures, [1] the others were written by third parties for special purposes, e.g. COSIMIR Freund *et al.* (2001). Such standalone tools usually perform very accurate simulations of PLCs including their time characteristics, but ordinarily, they can be embedded into another program only by limited or by complicated non-standard ways.

Before discussing the emulation itself, we first consider source codes of PLCs. The part 3 of IEC1131 standard, IEC (1993), defines a suite of programming languages recommended to be used with PLCs:

(1) Structured text — textual language with PASCAL like syntax and functionality;
(2) Instruction list — language resembling a typical assembler;
(3) (a) Ladder diagram — graphical language that appear as a schematics of the relay diagram; and
    (b) Function block diagram — graphical language resembling a logical schematics.

The structured text could the best source for emulations, unfortunately, few PLCs offer it. In contrast, the both graphical languages, which are implemented in a great number of PLC types almost exactly according to IEC 1131 specifications, only visualize internal PLC codes. Thus an instruction list ordinarily remains only one source for any PLC emulation.

Although many PLC types exist with various configurations, the number of their different instruction lists is much smaller due to backward compatibility of their software. For instance, Allen-Bradley's PLCs from families PLC-5, SLC-500, MicroLogix, and ControlLogix utilize similar in-

[1] For instance, Allen-Bradley sells emulators for its PLCs, but they only simulate PLC processors, so appropriate RSLogix development environments are also necessary for monitoring programs downloaded into emulated PLCs.

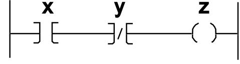

Fig. 1. Assignment $z = x \wedge \neg y$ in ladder diagram

struction lists, characterized in a lot of common features, which can be considered as belonging to one 'language group'. The rung of PLC ladder diagram depicted in Figure 1 corresponds to the source code "*XIC* x *XIO* y *OTE* z" in all mentioned PLC types.

If we define x, y, z variables and write proper emulation functions for all used PLC instructions, the rung above can be emulated by calling three functions, for instance: "*FnXIC(x); FnXIO(y); FnOTE(z);*". Such approach also allows modeling execution times of PLC instructions (if required) but it leads to long programs with many excessive calls, although the rung above can be emulated by one assignment $z = x \wedge \neg y$.

Assignments can emulate only pure control algorithm, but it is all we really need in many cases, and besides they are portable across whole range of programmable tools as a low cost emulation.

We based the conversion of PLC programs on the theory of the transfer sets that was originally designed for APLCTRANS algorithm Šusta (2003).

The emulation does not require the composition of whole PLC program into one automaton — every PLC rung (or PLC program block, respectively) can be converted into a separated statement. Therefore, it is possible to process a wider range of operations over APLCTRANS. [2]

2. OVERVIEW OF TRANSFER SETS

Here, we outline the transfer set theory adapted for a PLC emulation. In short, the theory formalizes concurrent evaluations of several expressions all at once. Suppose having variables x, y and C language assignments "$x = 2{*}x$; $y = x{+}1$;". Their classical consequent evaluation yields "$y = 2*x+1$;" for y variable, but their concurrent evaluation (utilized by the transfer sets) gives "$y = x{+}1$;" because x, y variables were assigned after evaluating the both expressions, thus the result corresponds to the program: "$temp_x = 2*x$; $temp_y = x+1$; $x = temp_x$; $y = temp_y$;" where $temp_x, temp_y$ are some temporary variables.

Transfer sets transparently specify complex transfer operations with program states. For instance, "push x" on an evaluation stack e_1, e_2, e_3 corresponds to three concurrent assignments "$e_1 = x$; $e_2 = e_1$; $e_3 = e_2$;" — notice that they may be listed in any order.

To reduce the size of this paper, we only present a rough definition of PLC storage and we replace the exact syntax of t-assignment expressions by the assumption of their similarity to C language, including ? : construction for the conditional assignments, but omitting pointers, ++ and - - operators. We hope that these short cuts do not confuse readers.

The conversion requires unambiguous relation between variables and memory. Let R be a set of PLC variables. If a boolean variable $b \in R$ is mapped to $I:1/0$ bit address (the least significant bit of $I:0$ input word) and an integer variable $w \in R$ is mapped to $I:0$ word address then the both variables share $I:1/0$ bit. We exclude this case by defining PLC storage.

Definition 1. (PLC storage). Let S be a set variables of a PLC. We will suppose that some given mapping of S into PLC memory is always firmly associated with S. Let $x \in S$ be any variable. The value of x evaluated with respect to S and to its mapping will be denoted by $[\![x]\!]\,S$. S is called a *PLC storage* if $[\![x]\!]\,S$ does not depend on $[\![y]\!]\,S$ for all possible contents of S and for arbitrary variables $x, y \in S$ such that $x \neq y$.

Let us write $EXP(S)^+$ for the set of all meaningful expressions over a given PLC storage S, i.e., their evaluation is known. $EXP(S)^+$ is nonempty because it contains at least numerical constants. If two (possibly different) expressions satisfy $[\![exp_1]\!]\,S = [\![exp_2]\!]\,S$ for all contents of S, we will write $exp_1 \equiv exp_2$, otherwise $exp_1 \not\equiv exp_2$.

Definition 2. Let $exp \in EXP(S)^+$ be any expression. The domain of exp is defined as:

$$\mathrm{dom}(exp) \stackrel{df}{=} \{v_i \in S \mid v_i \text{ is used in } exp\}$$

Definition 3. (T-assignment). Let S be a finite non-empty PLC storage and $v = [\![exp]\!]\,S$ be the assignment of $exp \in EXP(S)^+$ value to $v \in S$ variable. We define:

$$\begin{aligned} \hat{v}[\![exp]\!] &\stackrel{df}{=} v = [\![exp]\!]\,S \\ \mathrm{dom}(\hat{v}[\![exp]\!]) &\stackrel{df}{=} \mathrm{dom}(exp) \\ \mathrm{co}(\hat{v}[\![exp]\!]) &\stackrel{df}{=} v \end{aligned}$$

[2] APLCTRANS (Abstract PLC Transformation) was created for the verification of PLC programs. It performs an associative composition of some subset of PLC instructions into mathematical formulas and converts PLC program without loops in linear time in the size of its source codes at the most cases, though the program has an exponential complexity of its execution time. It was also proved that a PLC program can be modeled by a automaton of Mealy's family if and only if its operations are expressible with the aid of the instructions that are allowed by APLCTRANS.

where $\hat{v}[\![exp]\!]$ is called a *t-assignment*, and notations $\mathrm{dom}(\hat{v}[\![exp]\!])$ and $\mathrm{co}(\hat{v}[\![exp]\!])$ stand for its *domain* and *codomain*. If $exp \equiv v$, then $\hat{v}[\![exp]\!]$ is called a *canonical t-assignment*. The set of all t-assignments for S variables is denoted by $\widehat{\mathcal{T}}(S)$.

We have labeled t-assignments according to variables, but hat-accented, i.e., x variable has $\hat{x}$ t-assignment. We will also hat-accent all further objects related to t-assignments and our momentary assumptions about t-assignments will be expressed by their following forms:

$\hat{x}[\![exp]\!]$ represents a fully defined t-assignment for x variable with *exp* expression,
$\hat{x}[\![x]\!]$ stands for the canonical t-assignment for x variable, and
$\hat{x}$ denotes any t-assignment for x variable with an arbitrary $exp \in EXP(S)^+$.

T-assignments can be primed or subscribed, so $\hat{x_i}$, $\hat{x_j}$, and $\hat{y}$ represent t-assignments for three (possibly different) variables x_i, x_j, and y. If we need to distinguish among several t-assignments for one identical variable, we will always write them in their full forms — symbols $\hat{x}[\![exp_1]\!]$ and $\hat{x}[\![exp_2]\!]$ stand for two (possibly different) t-assignments for one x variable. The equality of t-assignments is determined by belonging to the same variable and their equivalent expressions.

Definition 4. Let $\hat{x}[\![exp_x]\!], \hat{y}[\![exp_y]\!] \in \widehat{\mathcal{T}}(S)$ be two t-assignments. Binary relation $\hat{x} \mathrel{\widehat{=}} \hat{y}$ is defined as the concurrent satisfaction of two following conditions: $\mathrm{co}(\hat{x}) = \mathrm{co}(\hat{y})$ and $exp_x \equiv exp_y$. If $\widehat{=}$ relation is not satisfied, we will emphasize this fact by $\hat{x} \mathrel{\widehat{\neq}} \hat{y}$.

Lemma 5. Binary relation $\widehat{=}$ on set $\widehat{\mathcal{T}}(S)$ is an equivalence relation.

Definition 6. (Transfer Set). *A transfer set on S storage* of a PLC is any subset $\widehat{X} \subseteq \widehat{\mathcal{T}}(S)$ that satisfies for all $\hat{x_i}, \hat{x_j} \in \widehat{X}$ that $\mathrm{co}(\hat{x_i}) = \mathrm{co}(\hat{x_j})$ implies $i = j$. We denote the set of all transfer sets for S variables by $\widehat{\mathcal{S}}(S)$, i.e., $\widehat{X} \in \widehat{\mathcal{S}}(S)$.

In other words, any transfer set contains at most one transfer function for each variable in S. The composition of transfer sets is based on the *concurrent substitution* defined here as a mapping from variables in S to terms of $EXP(S)^+$.

Definition 7. Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set and $exp_{dest} \in EXP(S)^+$ be any expression. *Concurrent substitution* $\widehat{X} \rightsquigarrow exp_{dest}$ is defined as such operation whose result is logically equivalent to the expression obtained by these consecutive steps:

(1) For all $\hat{x_i}[\![exp_i]\!] \in \widehat{X}$:
while $x_i \in \mathrm{dom}(exp_{dest})$, this x_i occurrence in exp_{dest} is replaced by some not interchangeable reference to x_i, where x_i represents $\mathrm{co}(\hat{x_i}[\![exp_i]\!])$.
(2) For all $\hat{x_i}[\![exp_i]\!] \in \widehat{X}$:
while the result of the previous step (modified expression exp_{dest}) contains a reference to $x_i = \mathrm{co}(\hat{x_i}[\![exp_i]\!])$ then x_i reference is replaced by "(exp_i)" i.e., the expression of $\hat{x_i}[\![exp_i]\!]$ enclosed inside parentheses.

Example 8. Given concurrent substitution:
$$\{\hat{x}[\![x \wedge y]\!], \hat{y}[\![\neg x \wedge \neg y]\!]\} \rightsquigarrow \hat{c}[\![(x \vee y) \wedge x]\!]$$
Direct application of the first step described in the definition above yields
$$\hat{c}[\![(\underline{x} \vee \underline{y}) \wedge \underline{x}]\!]$$
where underlining emphasizes that we have replaced variables by some unique references to the t-assignments that are not be their identifiers. The second step yields
$$\hat{c}[\![((x \wedge y) \vee (\neg x \wedge \neg y)) \wedge (x \wedge y)]\!]$$
but another acceptable results are also
$$\hat{c}[\![(x \equiv y) \wedge (x \wedge y)]\!] \quad \text{or} \quad \hat{c}[\![x \wedge y]\!]$$
because all expressions in three last t-assignments are equivalent.

Definition 9. (Weak composition). A weak composition $\widehat{Z} = \widehat{X} \circ \widehat{Y}$ of two given transfer sets $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ is the transfer set $\widehat{Z} \in \widehat{\mathcal{S}}(S), |\widehat{Z}| = |\widehat{X}|$, with t-assignments $\hat{x_i}[\![exp_{z,i}]\!] \in \widehat{Z}$ that are constructed for all $\hat{x_i}[\![exp_{x,i}]\!] \in \widehat{X}$ as:
$$\hat{x_i}[\![exp_{z,i}]\!] = \hat{x_i}\left[\!\!\left[\widehat{Y} \rightsquigarrow exp_{x,i}\right]\!\!\right]$$
where $i \in I$, $|I| = |\widehat{X}|, I = \{1, 2, \ldots, |I|\}$.

Example 10. Let $S = \{b, x, y, z\}$ be a PLC storage then the week composition of two given transfer sets is:

$$\begin{aligned}\widehat{C}_\circ &= \widehat{C}_2 \circ \widehat{C}_1 \\ &= \{\hat{x}[\![x * -y]\!]\} \circ \{\hat{x}[\![z/y]\!], \hat{y}[\![b\ ?\ y : z]\!]\} \\ &= \{\hat{x}[\![(z/y) * -(b\ ?\ y : z)]\!]\}\end{aligned} \tag{1}$$

Lemma 11. The weak composition $\circ$ is not associative on $\widehat{\mathcal{S}}(S)$.

Weak composition $\widehat{C}_2 \circ \widehat{C}_1$ can be modified to associative one, which we denote by $\circledcirc$, if we extend the leftmost transfer set $\widehat{C}_2$ before any composition by adding canonical t-assignments for all S variables, whose t-assignments are missing in the transfer sets. Let us write $\uparrow S$ for the extension operator described above, then $\widehat{C}'_2 = \widehat{C}_2 \uparrow S$ always satisfies $|\widehat{C}'_2| = |S|$.

After the composition, all canonical t-assignments are removed by the compression operator $\downarrow$, so the

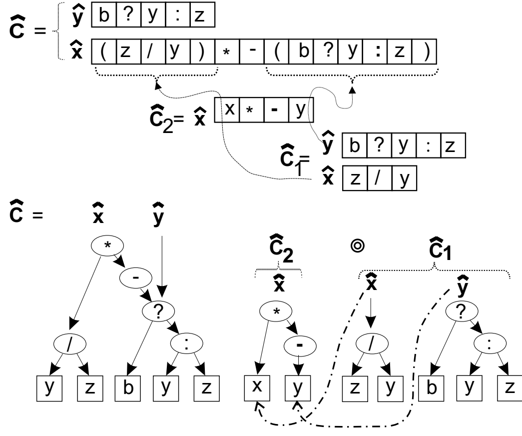

Fig. 2. Array versus graph implementation

associative compositions is defined as:

$$\widehat{C}_2 \circledcirc \widehat{C}_1 = ((\widehat{C}_2 \uparrow S) \circ \widehat{C}_1) \downarrow$$

The exact definitions are not too complex, but they need many auxiliary specifications. Thus we replace them by the following example. Readers may find the definitions including proofs and further details in Šusta (2003).

Example 12. We take PLC storage $S = \{b, x, y, z\}$ and transfer sets from Example 10.

$$\begin{aligned}
\widehat{C} = \widehat{C}_2 \circledcirc \widehat{C}_1 &= \Big((\widehat{C}_2 \uparrow S) \circ \widehat{C}_1\Big)\downarrow \\
&= \Big((\widehat{C}_2 \cup \{\hat{b}[\![b]\!], \hat{y}[\![y]\!], \hat{z}[\![z]\!]\}) \circ \widehat{C}_1\Big)\downarrow \\
&= \Big(\Big\{\hat{b}[\![b]\!], \hat{x}[\![x * -y]\!], \hat{y}[\![y]\!], \hat{z}[\![z]\!]\Big\} \circ \widehat{C}_1)\Big)\downarrow \\
&= \Big(\widehat{C}_\circ \cup \Big\{\hat{b}[\![b]\!], \hat{y}[\![b\ ?\ y : z]\!], \hat{z}[\![z]\!]\Big\}\Big)\downarrow \\
&= \{\ \hat{x}[\![(z/y) * -(b\ ?\ y : z)]\!], \hat{y}[\![b\ ?\ y : z]\!]\ \} \quad (2)
\end{aligned}$$

Notice that $\widehat{C}$ in Equation 2 differs from $\widehat{C}_\circ$ from Equation 1 only by the presence of $\hat{y}$ t-assignment.

2.1 Implementation of transfer sets

When transfer sets are stored as numeric arrays, the top part of Figure 2, each variable is coded as a unique number with the size according to requisite number of variables. The arrays have very simple implementation, but drawback in the multiplication of expressions. Each composition replaces every occurrences of one variable in the expressions by new string, which gradually increases the sizes of arrays.

The second proposed implementation utilizes the structure similar to Binary expression diagrams (BEDs), e.g. Andersen and Hulgaard (1997); Hulgaard *et al.* (1999), where logical subexpressions are not repeated, but shared, bottom of Figure 2.

Nowadays transfer sets are implemented only with the aid of byte arrays. New graph based version is under development and is expected to be available by the end of 2004.

3. CONVERSION OF PLC PROGRAM

This sections describes the steps necessary for converting a PLC program. First, we outline the conversion of PLC instructions to transfer sets, then we will illustrate the method by an example.

The simplest way of the conversion represents composing each rung, or a program block respectively, to one transfer set $\widehat{C}_i$, for $i = 1$ to $count_of_rungs$, and converting $\widehat{C}_i$ into programming statements of required language. Because one transfer set describes the concurrent assignments, we must convert t-assignments to programming assignments with the aid of temporary variables outlined at the beginning of Section 2 on page 2.

The composition is very fast operation, so we can also try composing two or more following $\widehat{C}_i$ one transfer set and then we select combination with lesser size of final source code.

The transfer sets for basic PLC instructions are listed in Šusta (2003) for some PLCs of Rockwell Automation (Allen-Bradley) and Siemens. These PLCs have instructions lists, where conditions are stored in f_{reg} boolean register and rung is evaluated with the aid of a boolean evaluation stack with limited depth.

Some transfer sets for PLC instructions are also presented in the example in Section 4.

3.1 Conversion of arithmetic instructions

Arithmetic instructions of PLC correspond to a conditional assignment $\hat{x}[\![f_{reg}?exp : x]\!]$, where exp describes the operation and $x \in S$ specifies a destination address. The assignment is evaluated as "if(f_{reg}) $x = exp$; else $x = x$;" imaginary program. In the case above, we may also remove else condition. For instance, ADD y 1 x instruction of PLC, which assigns $y + 1$ to x variable, is represented by $\{\ \hat{x}[\![f_{reg}\ ?\ y + 1\ :\ x]\!]\ \}$ transfer set, which corresponds to the statement: "if(f_{reg}) $x = y + 1$;". If f_{reg} was always true, i.e., the instruction had no condition then it would be represented by $\{\ \hat{x}[\![y + 1]\!]\ \}$ transfer set.

3.2 Side-effects of instructions

Side-effects mean that a PLC instruction alters its actual arguments or changes other variables.

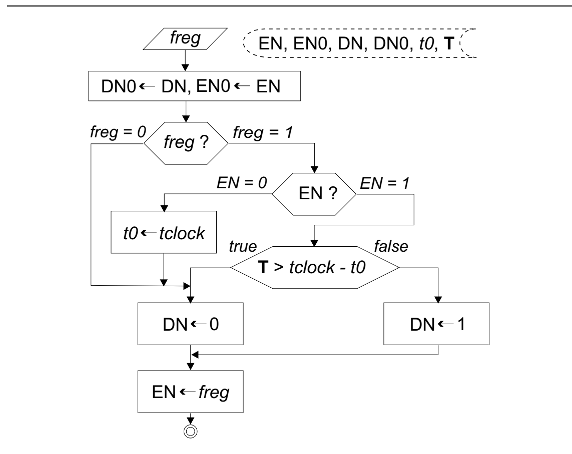

Fig. 3. Diagram of on-delay timer subroutine

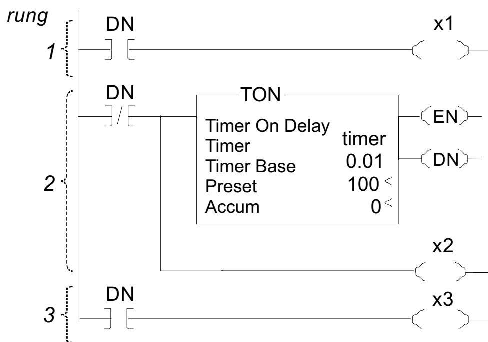

Fig. 4. Side-effects of on-delay timer (TON)

Instruction with side-effects cannot be normally converted into expressions where any of the other operands of the expression would be affected. They should be processed by special way.

Figure 4 shows PLC ladder diagram with on-delay timer TON. Its behavior can be emulated by the timer subroutine in Figure 3. The subroutine takes an evaluated input condition f_{reg} as its argument. It sets *DN* (done) and *EN* (enabled) output bits, which are tested by PLC instructions. *DN0* and *EN0* are their memories utilized for dealing with side-effects, $t0$ remembers the beginning of timing and it is loaded from a system *tclock* timer, and T is required the length of time.

The ladder diagram in Figure 4 can be converted by several ways: into one block by composing all rungs 1–3 into one transfer set; or into two blocks (rung pairs 1–2 and 3, or 1 and 2–3); or into three blocks by expressing each rung as one transfer set; eventually into more blocks, if some instructions will be emulated by stand-alone transfer sets.

In all cases, the call of the timer subroutine is inserted only in front the *block of converted instructions*, among which is a *timer* instruction — we denote this block by *BCIT* in this subsection.

The values of $x1$ and $x2$ outputs depends on the state of *DN* bit before calling the timer subroutine and by contrast, $x3$ value depends on the state of *DN* bit after its call. Therefore, we must also satisfy this property inside of BCIT.

If all output bits of a timer are read only after executing the timer instruction inside BCIT in all cases, the timer instruction is replaced only by $\{\hat{\tau_1}[\![f_{reg}]\!]\}$ transfer set, where τ_1 is a unique variable of the address utilized by the timer. It will be the input argument of the timer subroutine. [3]

Otherwise, if some timer outputs are also read inside BCIT before the timer instruction, the timer instruction is replaced by $\{\hat{\tau_1}[\![f_{reg}]\!], \hat{\tau_{1e}}[\![1]\!]\}$ transfer set, where τ_{1e} is a unique auxiliary variable initialized to zero by adding $\{\hat{\tau_{1e}}[\![0]\!]\}$ operation before BCIT, and the addresses of all timer bits accessed inside of this BCIT are replaced by multiplexing according to the state of τ_{1e}. For instance, *DN* bit will be replaced by the operation:

$$((\tau_{1e} \wedge DN) \vee (\neg\tau_{1e} \wedge DN0))$$

Notice, that this is necessary only in BCIT. The switching is useless in all other converted blocks, because there will be always $\tau_{1e} = 0$. [4]

The same method must be performed for all operations with side-effects or unpredictable effects, as storing data into an indirect address. To assure their correct evaluation, they must be converted as separate blocks, if it is possible, otherwise the switching of their before-after content should be included. In extremely critical cases, such operations can be emulated as the step by the step procedure, i.e. as stand alone statements. Therefore, side effect PLC operations will always require human's supervising and their fully automatic conversion is an idea for further research.

4. EXAMPLE OF CONVERSION

Figure 4 depict a fragment of a PLC program for counting boxes with a simple time filter of input. The instruction list exported from SLC 5/03 processor is:

```
SOR XIC S:1/15 CLR N7:0   EOR
SOR XIC I:1.0/0
      BST  TON T4:0 0.01 90 0
      NXB  XIC T4:0/DN   OSR B3:0/0
           ADD N7:0 1 N7:0
      BND
EOR
```

[3] Each address of timer instruction must be utilized only once times in any PLC program.

[4] Reading timer bits before executing timer instruction usually means an ineffective coding because we add the delay of one scan. Such operations are normally utilized only in few special cases, 2nd rung in Figure 4 illustrates one of them — it represents a pulse generator.

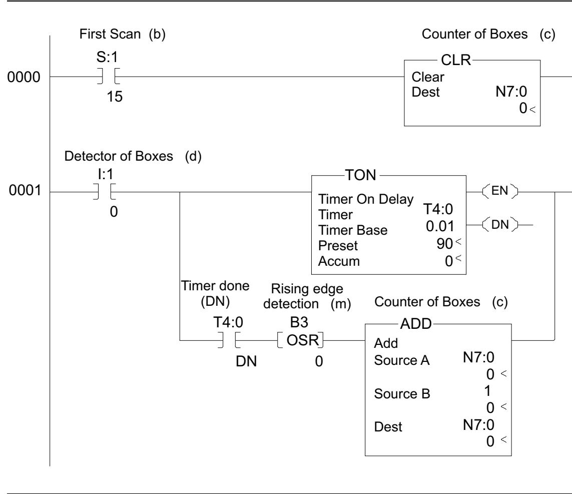

Fig. 5. Counter of boxes

If the detector I:1.0/0 gives signal longer than 0.9 second then N7:0 counter is incremented. The counter is initialized when PLC performs the first pass (bit S:1/15) after switching to run mode.

We first create PLC storage by assigning memory addresses to variables:

$b \Leftarrow$ S:1/15, $\quad c \Leftarrow$ N7:0,
$d \Leftarrow$ I:1.0/0, $\quad m \Leftarrow$ B3/0, and $DN \Leftarrow$ T4:0/DN,

to which we also add PLC flag register f_{reg} and PLC boolean evaluation stack e_1, e_2, whose depth is enough for this program. The last variable τ_1, corresponds to the input condition on the timer T4:0. It yields $S = \{b, c, d, m, DN, \tau_1, f_{reg}, e_1, e_2\}$ storage. Because our program does not access timer bits before evaluation of time instruction, we utilize simple emulation of timer and create transfer sets for the used instructions:

$$\begin{array}{ll}
\text{SOR, EOR} & \Rightarrow \{\hat{f_{reg}}[\![1]\!], \hat{e_1}[\![0]\!], \hat{e_2}[\![0]\!]\} \\
\text{BST} & \Rightarrow \{\hat{e_1}[\![0]\!], \hat{e_2}[\![f_{reg}]\!]\} \\
\text{NXB} & \Rightarrow \{\hat{f_{reg}}[\![e_2]\!], \hat{e_1}[\![f_{reg} \vee e_1]\!]\} \\
\text{BND} & \Rightarrow \{\hat{f_{reg}}[\![f_{reg} \vee e_1]\!], \hat{e_1}[\![0]\!], \hat{e_2}[\![0]\!]\} \\
\text{XIC x} & \Rightarrow \{\hat{f_{reg}}[\![f_{reg} \wedge x]\!]\} \\
\text{OSR m} & \Rightarrow \{\hat{f_{reg}}[\![f_{reg} \wedge \neg m]\!], \hat{m}[\![f_{reg}]\!]\} \\
\text{TON} & \Rightarrow \{\hat{\tau_1}[\![f_{reg}]\!]\} \\
\text{CLR c} & \Rightarrow \{\hat{c}[\![f_{reg} ? 0 : c]\!]\} \\
\text{ADD c 1 c} & \Rightarrow \{\hat{c}[\![f_{reg} ? c+1 : c]\!]\}
\end{array}$$

If we compose each rung separately, we obtain after removing uninterested f_{reg}, e_1 and e_2 that are initialized by EOR instruction:

$$\widehat{R}_1 = \{\hat{c}[\![\ b\ ?\ 0\ :\ c\ ]\!]\}$$

$$\widehat{R}_2 = \left\{ \begin{array}{l} \hat{c}[\![\ d \wedge DN \wedge \neg m\ ?\ c+1\ :\ c\ ]\!], \\ \hat{\tau_1}[\![d]\!], \hat{m}[\![d \wedge DN]\!] \end{array} \right\}$$

The transfer sets correspond to C language statements where the timer subroutine is called with d argument, i.e., $\hat{\tau_1}[\![d]\!]$ transfer set:

```
if(b) c=0;                    // 1st block
ExecuteTimer(d);              // 2nd block (BCIT)
m_temp = d && DN;
if(d && DN && !m) c_temp=c+1;
m=m_temp; c=c_temp
```

Finally we test composing the both rung into one block. It yields:

$$\widehat{R}_2 \circledcirc \widehat{R}_1 = \left\{ \begin{array}{l} \hat{c}\left[\!\!\left[\begin{array}{r} d \wedge DN \wedge \neg m\ ?\ (b\,?\,1\ :\ c+1) \\ :\ (b\,?\,0\ :\ c) \end{array} \right]\!\!\right], \\ \hat{t_e}[\![d]\!],\ \hat{m}[\![d \wedge DN]\!] \end{array} \right\}$$

This program evidently leads to longer source code, so the separate conversion is more efficient.

5. CONCLUSION

The presented method converts a PLC program to programming statements, which can be used either for a low cost emulation of the program or as auxiliary tool when a debugged PLC program is moved to another cheaper hardware, for instance to PC computer.

There are many possibility for future research and many lines of further development suggest themselves, for example optimization of producing final code to reduce requirements for temporary variables. An interesting approach also represents utilizing some faster analogy of the described PLC emulation (for instance with a partial aid of binary decision diagrams) for dynamic verification of PLC programs or for searching their state space.

REFERENCES

Andersen, Henrik Reif and Henrik Hulgaard (1997). Boolean expression diagrams. In: *LICS, IEEE Symposium on Logic in Computer Science.*

Freund, E., A. Hypki, F. Heinze and R. Bauer (2001). COSIMIR PLC - 3D simulation of PLC programs. In: *Proceedings of the 6th IFAC Symposium on Cost Oriented Automation, Berlin.*

Hulgaard, Henrik, Poul Frederick Williams and Henrik Reif Andersen (1999). Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design, 18(7).*

IEC (1993). *International Standard 1131, Programmable Controllers. Part 3: Programming Languages.* International Electrotechnical Commission.

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