A new Matlab coder for generating Structured Text Language from matrix expression for PLC and PAC controllers

Mariusz Buciakowski, Piotr Witczak
Institute of Control and Computation Engineering, University of Zielona Góra, ul. Podgórna 50, 65–246 Zielona Góra, Poland.
E-mail: {M.Buciakowski,P.Witczak}@issi.uz.zgora.pl

Abstract. This paper presents a new Matlab toolbox for synthesis of Structured Text (ST) code for Programmable Logic Controllers (PLC) and Programmable Automation Controller (PAC). This tool can directly generate IEC 61131-3 Structured Text Language from Matlab script for selected Integrated Development Environments (IDEs). The generated code can be verified and compared with the results obtained with Matlab simulation. After this, generated code can be used in IDEs, compiled and uploaded to a PLC or PAC controller for final verification. This approach leaves all available Matlab toolboxes for programmers use, thus allowing fast and easy synthesis developed algorithms.

1. Introduction
Nowadays, when industrial processes, as well as systems that controls them, become more and more complex, new challenges emerges. The most obvious one, directly related to the safety and security, but also to cost minimization is the reliability of used software and hardware. Secondly, the time of software implementation and its deployment to production hardware affects the time-to-delivery and total production costs. One of the possible solutions that can overcome at least part of the mentioned issues is automatic synthesis of standardized IEC 61131-3 Structured Text [7] code for Programmable Logic Controllers (PLC) and Programmable Automation Controller (PAC) controllers. Many researchers focuses on the different aspects of this area. Some of them proposes component-based modeling [16], others on the code reusability [1, 5], function-block-based modeling [6], generative programming [4] or sequential circuit synthesis [9]. While a modeling can take different forms, all of them leads to a code synthesis. All mentioned approaches have their potential advantages and drawbacks. It is enough to mention code reusability and function-block-based solution. The first one proposes the import/export mechanisms to reuse once designed and written code between platforms. While the idea is quite simple, the realization with numerous hardware platforms may not be so trivial as it is common that some manufacturers treats non-standardised elements on their own (i.e. totally different ST variable definition in IDE, Profic Machine Edytion IDE, Step7 IDE etc.). Additionally solution does not offer the homogenous, safe verification environment. Leaving IDE in the middle of the binary code generation, allows hardware-dependent optimization (being a part of software delivered by controller manufacturer). The function-block-based is
on the other side limited by function-blocks, and complicated programs become also graphically complex (cf. [6]). The last one, however, offers simulation perspectives. Due to the authors knowledge there is in fact a gap in the area of code synthesis for PLC and PAC. Especially, when modern controllers are capable of handling complex arithmetic operations in real-time. This paper proposes a solution for model-based code-synthesis in terms of mathematical modeling. The toolbox described in later sections is capable of Matlab-to-ST code conversion, preserving structure of mathematical components, such as matrices and vectors. Therefore, a model-based controller or filters can be built relatively fast. They can be also easily verified before uploading them into the device. This phenomena can be quite useful when on a PLC or PAC instead of classical industrial control scheme, the modern control would be implemented. Also thanks to the direct use of Matlab it is possible to use such tools as neural networks toolbox, LMIs [3, 12], P-matrices [8] and many others (on the high quality level). The recent works in the area of Fault Detection and Isolation (FDI) [21, 20, 11, 15] as well as Fault-Tolerant Control (FTC) clearly show a new direction for the industrial evolution. If one would compare any currently used control approach with any advanced FTC the superiority of the last one becomes more than obvious [18, 22, 13, 17, 19, 10]. However the main obstacles are - the lack of trust form the industry in new techniques and the lack of tools for simple and fast implementation of advanced modes into PLC/PAC.

The paper is organised as follows. Section 2 is devoted to the structure of the framework, as well as its general capabilities. Subsequently Section 3 shows some examples. Finally Section 4 summarizes the paper.

2. Toolbox structure

The main objective of this section is to present structure of toolbox, functionalities and the generic functions of the framework. The Toolbox for generating ST language was implemented using object–oriented programming techniques in Matlab. The use of this programming techniques allows to implement interface for the toolbox in an efficient way. Also it allows fast expansion by providing a properly built interfaces. The number of parameters that should be passed to the toolbox is as small as possible, providing a high user experience and usability. All of this allows comfortable usage, with only basic knowledge about Matlab and simple extension to any unimplemented ST version. Apart of above the framework consists of two main classes, from which one of them directly implements the conversion interface. The second one is a storage class based on the factory pattern [2]. Therefore, it fulfill two roles, one - stores all components for language conversion and code generation, secondly builds convertible objects. The toolbox has few main functionalities. The first functionality allows to select IDE. To do so user should use the following command:

```matlab
cs = ConversionStorage('Cscape');
```

Currently the toolbox supports the following IDE name:

(i) 'Cscape' – Cscape IDE for Horner APG PLC,
(ii) 'PME' – Proficy Machine Edition IDE for GE PAC controllers,
(iii) 'Step7' – Step7 for Siemens Simatic S7 300/400 controllers.

The code generation procedure requires the object `ConversionStorage` to be made, as one of its methods invokes the language conversion. The toolbox check name of IDE. If user use IDE name that not supported now, the error will return. The numbers for supported IDE will grow. The storage needs to consist of all variables used in the final program, therefore each variable from the workspace needs to be registered in the instance `cs`

```matlab
cs = ConversionStorage(IDE);
```
Furthermore, user can check weather variables are properly added to the working instance of `ConversionStorage`:

```java
sc.show();
```

The main functionality is the conversion method, that needs to be invoked with the parameter that defines the model to be converted:

```java
sc.convert(x);
```

where `x` needs to be defined Workspace char variable (see Section 3). Last but not least is the method that allows user to save code in a file:

```java
sc.drop2File('text.txt');
```

3. Examples

The objective of this section is to present examples of the generation code for selected IDE. Initial example provides an insight to the basic usage, while subsequent provides more detailed and advanced options. The first examples illustrate full code for setting initial values of variables and for calculating expression. The rest of examples show only code for calculating expression to present advantages for use this toolbox.

Note that the variable needs to be firstly created in a Matlab Workspace. For example let us define two simple variables

```java
A = [1 2; 3 4];
b = [4; 3];
```

one matrix and one column vector

```latex
\begin{bmatrix}
  1 & 2 \\
  3 & 4
\end{bmatrix}, \begin{bmatrix}
  4 \\
  3
\end{bmatrix}
```

and expression that multiplies two above matrices

```latex
C = A*b;
```

The above will simply create a `C` variable in Matlab Workspace. In order to create a Toolbox valid variable for conversion, one simply changes the expression into a set of `char`

```java
C = 'A*b';
```

and then converts it to ST by invoking

```java
sc.convert(C);
```

On this level it is worth to mention that the type of variable is automatically detected, as well as type of The result. The converted type of variable depends on selected IDE however toolbox supports only `uint8, int16, uint16, int32, double, single` Matlab type of variable. The above allows to easily translate variable passed to the toolbox into a common variable type supported by PLCs and PACs. Subsequently the resulting code should be stored into a file

```java
sc.drop2File('scape.txt');
```

The file name is used for create *.txt file which will contain ST code. The full code for example (2) is
\[ A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}; \]
\[ b = \begin{bmatrix} 4 \\ 3 \end{bmatrix}; \]
\[ sc = \text{ConversionStorage('Scape');} \]
\[ sc\text{.addVariable(A);} \]
\[ sc\text{.addVariable(b);} \]
\[ sc\text{.show();} \]
\[ C = 'A*b'; \]
\[ sc\text{.convert(C);} \]
\[ sc\text{.drop2File('scape.txt')} \]

The ST code is
\[ A[0,0] := 0; \]
\[ A[0,1] := 1; \]
\[ A[1,0] := 2; \]
\[ A[1,1] := 3; \]
\[ b[0] := 3; \]
\[ b[1] := 2; \]
\[ C[0] := A[0,0] \times b[0] + A[0,1] \times b[1]; \]
\[ C[1] := A[1,0] \times b[0] + A[1,1] \times b[1]; \]

**EXAMPLE 1**: The first examples base on expression

\[ C = A + B \] (3)

where

\[ A = \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix} \]
\[ B = \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix} \] (4)

and the respective Matlab code

\[ A = \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix}; \]
\[ B = \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix}; \]
\[ sc = \text{ConversionStorage('PME');} \% \text{Step7} \]
\[ sc\text{.addVariable(A);} \]
\[ sc\text{.addVariable(B);} \]
\[ sc\text{.show();} \]
\[ C = 'A+B'; \]
\[ sc\text{.convert(C);} \]
\[ sc\text{.drop2File('pme.txt')} \]

*Listing. 1* and *Listing. 2* show full code for PME and Step7 respectively. The code also it contains instructions to set initial value for each variable using in expression. In case of PME, the initial value can be set in the same block where the main code is used to compute expression or in a different block. It depends on the Programmer preferences however, the initial values can be set once, using information about first cycle in PLC or PAC. For Step7, toolbox generate full Function Block 1. The number of Function Block is default set to 1. The generated block can be compiled and sent to the controller.

**EXAMPLE 2**: The second example shows ST code for expression

\[ z_k = A\hat{x}_k + Bu_k + K(y_k - C\hat{x}_k). \] (5)

The above expression describe Luenberger Observer and it is commonly used in control theory. In (5) \( u, y, \) and \( \hat{x} \) are input signal. The codes example (the values for each matrices is omitted)
Listing. 1 ST code for Proficy Machine Edition IDE for example 1

```
A[0,0] := 2.000000;
A[0,1] := 3.000000;
A[1,0] := 2.000000;
A[1,1] := 3.000000;
B[0,0] := 2.000000;
B[0,1] := 3.000000;
B[1,0] := 2.000000;
B[1,1] := 3.000000;
C[0,0]:=A[0,0]+B[0,0];
C[0,1]:=A[0,1]+B[0,1];
C[1,0]:=A[1,0]+B[1,0];
C[1,1]:=A[1,1]+B[1,1];
```

Listing. 2 ST code for Step7 IDE for example 1

```
FUNCTION_BLOCK FB1
VAR
  A : ARRAY[0..1,0..1] OF REAL :=
      2.000000,3.000000,
      2.000000,3.000000;
  B : ARRAY[0..1,0..1] OF REAL :=
      2.000000,3.000000,
      2.000000,3.000000;
  C : ARRAY[0..1,0..1] OF REAL;
END_VAR
C[0,0]:=A[0,0]+B[0,0];
C[0,1]:=A[0,1]+B[0,1];
C[1,0]:=A[1,0]+B[1,0];
C[1,1]:=A[1,1]+B[1,1];
END_FUNCTION_BLOCK
```

```
sc = ConversionStorage('PME'); % Step7
sc.addVariable(A);
sc.addVariable(x);
sc.addVariable(B);
sc.addVariable(u);
sc.addVariable(K);
sc.addVariable(y);
sc.addVariable(C);
z = 'A*x+B*u+K*(y-C*x)';
sc.convert(z)
sc.drop2File('pme.txt')
```
show Listing. 3 and Listing. 4 show full code for PME and Step7 respectively. ST code in Listing. 3 is similar for code in Listing. 1. In this example expression includes input variables, therefore code for Step7 IDE contains CASE statement for set initial value for variables $u$, $y$, and $\dot{x}$. This statement can be also use for reset variables to initial condition.

**Listing. 3** ST code for Proficy Machine Edition IDE for example 2

\[
\begin{align*}
A[0,0] &:= 32.000000; \\
A[0,1] &:= 22.000000; \\
A[1,0] &:= 23.000000; \\
A[1,1] &:= 85.000000; \\
x[0] &:= 40.000000; \\
x[1] &:= 68.000000; \\
B[0] &:= 83.000000; \\
B[1] &:= 70.000000; \\
u &:= 92.000000; \\
K[0] &:= 94.000000; \\
K[1] &:= 38.000000; \\
y &:= 25.000000; \\
C[0] &:= 96.000000; \\
C[1] &:= 55.000000; \\
z[0,0] &:= 0; \\
z[0,1] &:= 0; \\
z[0,0] := & B[0] \ast u - K[0] \ast (-y + C[0] \ast x[0] + C[0] \ast x[1]) + A[0,0] \ast x[0] + A[0,1] \ast x[1]; \\
z[1,0] := & B[1] \ast u - K[1] \ast (-y + C[0] \ast x[0] + C[1] \ast x[1]) + A[1,0] \ast x[0] + A[1,1] \ast x[1];
\end{align*}
\]

**EXAMPLE 3**: This example shows how generate code for more advanced Matlab functions. In the following example we want to generate code based on the Jacobian of function:

\[
s = \text{jacobian}(2xysin(z) + 3xsin(y)cos(z))
\]

The full code for example

\[
s = '\text{jacobian}(2\ast X \ast Y \ast \sin(Z) + 3\ast X \ast \sin(Y) \ast \cos(Z), [X, Y, Z])';
\]

\[
sc.addVariable(X);
sc.addVariable(Y);
sc.addVariable(Z);
sc.convert(s)
\]

**Listing. 5 and Listing. 6** show ST code. From listings, it can be seen that the toolbox allows to generate code for more advanced Matlab functions.

**EXAMPLE 4**: The next example shows code for following dynamic linear discrete-time system

\[
x_{k+1} = Ax_k + Bu_k
\]

where $A$, $B$, $u$ are constant matrices of appropriate dimensions. Similarly to the previous examples Matlab code is
Listing. 4 ST code for Step7 IDE for example 2

FUNCTION_BLOCK FB1
VAR_INPUT
  INIT_VAR : INT;
END_VAR
VAR
  A : ARRAY[0..1,0..1] OF REAL :=
    32.000000,22.000000,
    23.000000,85.000000;
  x : ARRAY[0..1] OF REAL;
  B : ARRAY[0..1] OF REAL :=
    83.000000,70.000000;
  u : REAL;
  K : ARRAY[0..1] OF REAL :=
    94.000000,38.000000;
  y : REAL;
  C : ARRAY[0..1] OF REAL :=
    96.000000,55.000000;
  z : ARRAY[0..1,0..1] OF REAL;
END_VAR
CASE INIT_VAR OF
  1 : u :=0;
      y :=0;
      x[0]:=0;
      x[1]:=0;
      z[0,0]:=0;
      z[0,1]:=0;
ELSE:
  z[0,0]:=B[0]*u-K[0]*(-y+C[0]*x[0]+C[1]*x[1])+A[0,0]*x[0]+A[0,1]*x[1];
  z[0,1]:=B[1]*u-K[1]*(-y+C[0]*x[0]+C[1]*x[1])+A[1,0]*x[0]+A[1,1]*x[1];
END_CASE;
END_FUNCTION_BLOCK

Listing. 5 ST code for Proficy Machine Edition IDE for example 3

X := 0;
Y := 0;
Z := 0;
s[0,0]:=Y*sin(Z)*2.0+cos(Z)*sin(Y)*3.0;
s[0,1]:=X*sin(Z)*2.0*X*cos(Y)*cos(Z)*3.0;
s[0,2]:=X*sin(Y)*sin(Z)*-3.0+X*Y*cos(Z)*2.0;
Listing. 6 ST code for Step7 IDE for example 3

FUNCTION_BLOCK FB1
VAR_INPUT
   INIT_VAR : INT;
END_VAR
VAR
   X : REAL;
   Y : REAL;
   Z : REAL;
   s : ARRAY[0..2,0..1] OF REAL;
END_VAR
CASE INIT_VAR OF
   1 : X :=0;
   Y :=0;
   Z :=0;
ELSE:
   s[0,0]:=Y*sin(Z)*2.0+cos(Z)*sin(Y)*3.0;
   s[0,1]:=X*sin(Z)*2.0+X*cos(Y)*cos(Z)*3.0;
   s[0,2]:=X*sin(Y)*sin(Z)*-3.0+X*Y*cos(Z)*2.0;
END_CASE;
END_FUNCTION_BLOCK

x = 'A*x+B*u';
s = ConvolutionStorage('PME'); % or Step7
s.addVariable(A);
s.addVariable(B);
s.addVariable(u);
s.convert(x);

Listing. 7 ST code for Proficy Machine Edition IDE for example 4

A[0,0] := 40.000000;
A[0,1] := 82.000000;
A[1,0] := 20.000000;
A[1,1] := 58.000000;
B[0] := 50.000000;
B[1] := 86.000000;
u := 20.000000;

x[0]:=A[0,0]*x[0]+A[0,1]*x[1]+B[0]*u+B[1]*u;
x[1]:=A[1,0]*x[0]+A[1,1]*x[1]+B[0]*u+B[1]*u;
Listing. 8 ST code for Step7 IDE for example 4

FUNCTION_BLOCK FB1
VAR
A : ARRAY[0..1,0..1] OF REAL :=
    40.000000,82.000000,20.000000,58.000000,
B : ARRAY[0..1] OF REAL := 50.000000,86.000000;
    u : REAL := 20.000000;
x : ARRAY[0..5,0..5] OF REAL;
END_VAR

    x[0]:=A[0,0]*x[0]+A[0,1]*x[1]+B[0]*u+B[1]*u;
    x[1]:=A[1,0]*x[0]+A[1,1]*x[1]+B[0]*u+B[1]*u;
END_FUNCTION_BLOCK

4. Concluding Remarks
The paper deals with the emerging and important issue of Structured Text code synthesis for PLC and PAC controllers. Authors proposes a Matlab based, model-based approach to automatic code generation. Thanks to Matlab powerful seamless toolboxes any demanding control task can be solved efficiently and subsequently tested within the encapsulated and safe environment. This procedure saves cost, time and equipment from damage. At the same time final product such as control scenario or observer structure can be converted into ST language and uploaded into the controller of given manufacturer. Siemens, General Electric or Horner makes no difference thanks to a interface-like object-oriented design of a framework. The Matlab toolbox build for this task can be easily scaled to a incoming needs. Furthermore, a future development of the toolbox is planed, e.g.to automatically handle gain scheduling from Linear Parameter-Varying [14] models or membership functions from Takagi-Sugeno [23] models.

5. Acknowledgments
Authors would like to express their sincere thanks to colleagues from the Institute for valuable hints and motivating discussions. The work was supported by the National Science Centre of Poland under grant: 2013/11/B/ST7/01110

Reference
[1] Tord Alenljung, Markus Skoldstam, Bengt Lennartson, and Knut Akesson. PLC-based implementation of process observation and fault detection for discrete event systems. In 2007 IEEE International Conference on Automation Science and Engineering, pages 207–212. IEEE, 2007.
[2] Andrei Alexandrescu. Modern C++ design: generic programming and design patterns applied. Addison-Wesley, 2001.
[3] S. Boyd, E. Feron, L. E. Ghaoui, and V. Balakrishnan. Linear Matrix Inequalities in System and Control Theory. Siam, Philadelphia, 1994.
[4] Daniel Côté, Richard St-Denis, and Sylvain Kerjean. Generative programming for programmable logic controllers. In 2005 IEEE Conference on Emerging Technologies and Factory Automation, volume 2, pages 8–pp. IEEE, 2005.
[5] E Estevez, M Marcos, E Irisarri, F Lopez, I Sarachaga, and A Burgos. A novel approach to attain the true reusability of the code between different PLC programming tools. In Factory Communication Systems, 2008. WFCs 2008. IEEE International Workshop on, pages 315–322. IEEE, 2008.
[6] Knut Guttel, Peter Weber, and Alexander Fay. Automatic generation of PLC code beyond the nominal sequence. In 2008 IEEE International Conference on Emerging Technologies and Factory Automation, pages 1277–1284. IEEE, 2008.
[7] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.

[8] B. Kolodziejczak and T. Szulc. Convex combinations of matrices. full rank characterization. *Linear algebra and its applications*, 287(1-3):215–222, 1999.

[9] T Krairojananan and S Suthapradit. A PLC program generator incorporating sequential circuit synthesis techniques. In *Circuits and Systems, 1998. IEEE APCCAS 1998. The 1998 IEEE Asia-Pacific Conference*, pages 399–402. IEEE, 1998.

[10] Marcel Luzar, Marcin Witczak, Mariusz Buciakowski, and Vicenç Puig. Robust $H_\infty$ actuator fault diagnosis and fault-tolerant control for a multi-tank system. In *2013 Conference on Control and Fault-Tolerant Systems (SysTol)*, pages 758–763. IEEE, 2013.

[11] Marcin Mrugalski, Marcel Luzar, Marcin Pazera, Marcin Witczak, and Christophe Aubrun. Neural network-based robust actuator fault diagnosis for a non-linear multi-tank system. *ISA transactions*, 61:318–328, 2016.

[12] Boris Polyak and Pavel Shecherbakov. Ellipsoidal approximations to attraction domains of linear systems with bounded control. In *American Control Conference*, 2009.

[13] V. Puig. Fault diagnosis and fault tolerant control using set-membership approaches: Application to real case studies. *International Journal of Applied Mathematics and Computer Science*, 20(4):619–635, 2010.

[14] Damiano Rotondo, Fatiha Nejjari, and Vicenc Puig. Robust quasi LPV model reference FTC of a quadrotor UAV subject to actuator faults. *International Journal of Applied Mathematics and Computer Science*, 25(1):7–22, 2015.

[15] Damiano Rotondo, Marcin Witczak, Vicenç Puig, Fatiha Nejjari, and Marcin Pazera. Robust unknown input observer for state and fault estimation in discrete-time takagi–sugeno systems. *International Journal of Systems Science*, 47(14):3409–3424, 2016.

[16] Jan Thieme and Hans-Michael Hanisch. Model-based generation of modular PLC code using iec61131 function blocks. In *Proceedings of the International Symposium on Industrial Electronics*, volume 1, pages 199–204, 2002.

[17] K.C. Veluvolu, M.Y Kim, and D. Lee. Nonlinear sliding mode high-gain observers for fault estimation. *International Journal of Systems Science*, 42(7):1065–1074, 2011.

[18] M. Witczak. *Fault Diagnosis and Fault-Tolerant Control Strategies for Non-linear Systems*. Springer-Verlag, Berlin-Heidelberg, 2014.

[19] Marcin Witczak, Mariusz Buciakowski, and Christophe Aubrun. Predictive actuator fault-tolerant control under ellipsoidal bounding. *International Journal of Adaptive Control and Signal Processing*, 30(2):375–392, 2016.

[20] Marcin Witczak, Mariusz Buciakowski, and Marcin Mrugalski. An $H_\infty$ approach to fault estimation of nonlinear systems: Application to one-link manipulator. In *Methods and Models in Automation and Robotics (MMAR), 2014 19th International Conference On*, pages 456–461. IEEE, 2014.

[21] Marcin Witczak, Mariusz Buciakowski, Vicenç Puig, Damiano Rotondo, and Fatiha Nejjari. An LMI approach to robust fault estimation for a class of nonlinear systems. *International Journal of Robust and Nonlinear Control*, 26(7):1530–1548, 2016.

[22] P. Witczak, M. Witczak, K. Patan, and R. Stetter. Design of robust predictive fault-tolerant control for takagi–sugeno fuzzy systems: Application to the twin-rotor system. In *Intelligent Control (ISIC), 2014 IEEE International Symposium on*, pages 1113–1118, Oct 2014.

[23] Piotr Witczak and Marcin Witczak. Towards robust fault-tolerant model predictive control with constraints for takagi–sugeno systems. In *Variable-Structure Approaches*, pages 309–333. Springer International Publishing, 2016.