Developing a Multiplatform Control Environment

Submitted: 15th July 2019; accepted: 20th December 2019

Dariusz Rzońca, Jan Sadolewski, Andrzej Stec, Zbigniew Świder, Bartosz Trybus, Leszek Trybus

DOI: 10.14313/JAMRIS/4-2019/40

Abstract:

IEC 61131-3 control environment is called multiplatform if source programs can be executed by various processors, beginning from 8-bit microcontrollers up to 32/64-bit efficient CPUs. This implies that virtual machine (VM), i.e. a software implemented processor, is used as runtime by the host CPU. The VM executes certain intermediate code into which IEC 61131-3 programs are compiled. Environment of this type called CPDev has been gradually developed by the authors over the last decade, beginning with initial report in this journal in 2008 [47]. However, technical implementations of its functionalities have not been described so far. This involves such matters as intermediate language, parametrization of the compiler and VM, multiproject runtime, translators of graphical languages, device-independent HMI, target platform and communication interfacing, which are presented in compact form in this paper. Some characteristic industrial implementations are indicated.

Keywords: *IEC 61131-3 environment, virtual machine, intermediate language, parametrized compiler, interfacing, HMI, industrial controllers*

1. Introduction

There is a common opinion among practising engineers and university staff that runtime engineering environments based on IEC 61131-3 standard [22] will remain state of industrial practice long into the next decade (e.g. [52], [56]). The standard defines five programming languages, i.e. textual IL, ST, graphic LD, FBD and mixed SFC, with time-triggered scan cycle execution. IL, LD and SFC are preferred in manufacturing based on PLCs, whereas general automation favours ST, FBD and also SFC. Edition 3.0 (2013) of the standard has introduced object-oriented programming by extending the original function block concept.

Open architecture for distributed control and automation focused on interoperability of the devices configured by multiple tools, with function blocks as reusable software components, is defined in another IEC 61499 control standard [23]. Runtime software is event-driven. Code measures have shown [17] that IEC 61499 is more appropriate for sequential controls, but IEC 61131-3 suits better general applications. However, due to insufficient support by commonly used tools and runtimes IEC 61499 has not been generally accepted by industry so far [52], although some successful applications particularly in energy sector have been reported (e.g. [10]).

Compilers of control languages are basic components of IEC 61131-3 tools. A compiler translates source program into machine code executed by runtime software of the controller processor. Three approaches to compilation are encountered in practice. In the first one, most common, IEC 61131-3 programs are directly translated into machine code. Execution is fast, so the approach is applied by leading manufacturers of control equipment, such as Siemens, ABB, Schneider, Rockwell, Omron and others. Some tools from independent software providers, namely CODESYS [1] (market leader) and LogicLab [2] also directly translate source programs into machine code. It is however a single platform solution since change of CPU requires a new compiler.

Contrary to the direct translation the second approach involves two steps. At first IEC 61131-3 programs are translated to C/C++ and then another tool generates the machine code. Academic open-source Beremiz [53] and independent GEB [16] apply this approach which due to common C/C++ tools suits multiple platforms, not just one as before. The approach is particularly suitable for education, however necessity of another C/C++ translation limits somewhat commercial applications.

In the third approach, source programs are compiled into binary code of a dedicated intermediate language executed by specially designed virtual machine (VM), i.e. a software processor. The VM is in fact the runtime program of the target controller. The approach does not need translation to C/C++ and may be applied to multiple platforms. It was originally introduced in ISaGRAF environment [21] with intermediate Target Independent Code. STRATON [9] (independent) also applies this approach, however details remain confidential. VMs are typical for academic solutions reviewed in the next section. Nevertheless, time efficiency of the approach is lower due to overhead imposed by the VM.

The other IEC 61499 standard is supported by several engineering tools of which nxtControl [38] used at academic institutions may be an example. Also ISaGRAF provides now IEC 61499 programming option.

The authors began developing an IEC 61131-3 environment over a decade ago encouraged by an equipment manufacturer and because of teaching needs. It was actually a next step from earlier works on multifunction controllers (e.g. [7]). Application of the first version of the environment called Control Program Developer (CPDev) was reported in this journal [47] (also shown at Automation 2008 Fair). The first ver-

sion compiled ST programs into intermediate Virtual Machine Assembler (VMASM) executed by runtime VM on AVR and MCS51 8-bit microcontrollers. It already involved certain provisions to facilitate porting to different hardware and interfacing communications.

Over the following years CPDev has been extended on 32/64-bit processors, augmented with translators of graphical languages, HMI design tool and multi-project runtime (initially applied in FPGA [18]). PhD research has been supported by Model Driven Development and unit-testing extensions [29], [27]. So far several Small and Medium Scale (SMES) manufacturers use the environment.

The early paper [47] and the others have focused on CPDev functionalities rather than on explaining how they have been created. Therefore the purpose of this paper is to present technical aspects of:

- portability of the compiler and VM to different CPUs
- implementation of interfaces for various target platforms
- translators of graphical languages
- structure of device-independent HMI tool
- data for binding variables to communication interfaces.

This paper is thus organized as follows. Next section reviews related work concerning IEC 61131-3 environments. Section 3 summarizes basic features of CPDev. Section 4 overviews VMASM language, parametrized compiler and presents example running throughout the paper. Architecture of the VM, implementation of target platform interfaces and multi-project runtime are described in Section 5. Section 6 presents translators of graphical languages and structure of the HMI tool. List of data required to bind variables to communication interfaces and some characteristic industrial implementations are shown in Section 7.

2. Related Work

An interest in virtual machines, i.e. abstract processors for specific languages implemented by software on particular hardware platforms has been increasing since Java Virtual Machine (JVM) appeared over 20 years ago [55], followed later by Common Language Runtime (CLR) for .NET framework [46]. In particular, most of ARM-based mobile phones implement JVM processors in Jazelle technology.

VM-based control solutions with the standard IL as the intermediate language appeared in literature in similar time as the early paper [47] (besides commercial ISaGRAF). VM of [58] was written in C and applied in 8-bit C8051 CPU. Whimori CDU [48] educational tool translated LD into IL. UniSim [11], although restricted to a few data types, provided graphical programming. IL was also used in [57] for VM implementation of ARM-based softcontroller.

More recently, IEC 61131-3 applications have been executed by CLR, i.e. a .NET VM [5]. Development of various educational tools continues. For example, WEB PLC Simulator [41] executes control algorithm and plant model written in ST. HT-PLC [26] runs LD programs on Arduino with Ethernet. Among relevant solutions in other areas one may mention security-oriented IoT environment [32] involving SIL intermediate language, C/C++/Java-to-SIL compiler and VM.

Scan cycle, important particularly in manufacturing, can be decreased by direct execution at a dedicated processor, as shown in [39] for a Toshiba CPU. Nowadays hardware processors are designed in FPGA technology, as demonstrated by early double processor for a PLC [6] and in [42], [35] for current multi-core solutions. Comparison of execution times for a few PLCs and FPGA has been presented in [18]. A number of FPGA designs can be found in the proceedings of PDES conferences (Programmable Devices and Embedded Systems).

Details of IEC 61131-3 compilers are typically not disclosed. As an exception the early paper [31] describes structure of the compiler, code optimization and support of logic operations by a programmable device. Nevertheless, one may generally expect that most of the compilers apply Static Single Assignment (SSA) syntax and corresponding translation described for instance in [8]. Occasionally semantic-driven translation can also be encountered [13]. SSA syntax has been recently applied for inference of types in dynamic languages [45], such as Python. Parametrization of GCC compiler due to limited resources has been presented in [44].

Activity on Vertex (AOV) graph transformation algorithms are typically used to translate LD diagrams into IL or ST [12], [20]. Direct translation of AOV into ST is applied in CPDev. More advanced Enhanced Data Flow Graphs from [36] can implement IL, LD and SFC programs.

Parametrized formal semantics for execution of SFC diagrams has been proposed in [4] to avoid ambiguities of IEC 61131-3 standard. Unsafe components of SFCs created using CODESYS semantics (also applied in CPDev) can be detected by static analysis [49].

Principles of HMI design for control applications have been described in [14]. German standard VDI/VDO 3699 [51] also covers this area. Future advanced solutions may involve 3D graphics and virtual reality [50]. Some features of industrial operator panels and in-vehicle infotainment systems [34] are clearly the same.

Implementation guidelines and technical details concerning industrial communications can be found for instance in [59]. Comparison of industrial and public networks taking into account control, standardization and dependability is discussed in [15].

3. CPDev Overview

The CPDev tool, initially providing just ST programming, supports now all IEC 61131-3 languages, HMI design, documentation generator and research-oriented extensions. Multi-project runtime for complex software is also available. Steps of control program processing are shown in Fig. 1. ST and IL programs are compiled into mnemonics of the intermedi-

ate VMASM. LD, FBD and SFC diagrams are translated to ST and then compiled. Assembler converts the mnemonics into executable binary code uploaded into VM. The VM is written in C, so may be implemented in various CPUs.

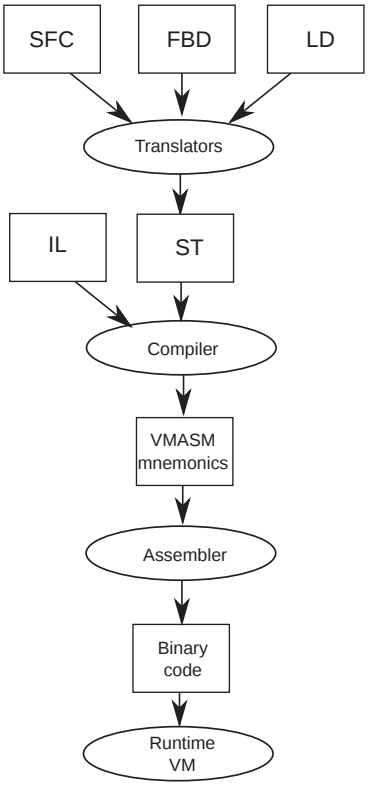


Fig. 1. Processing of control programs

Basic functionalities of CPDev are practically the same as in other control environments. Automatic connections provided by LD and FBD editors may be an exception (A* algorithm [19]). HMI tool applies components from device-independent libraries. Data for interfacing communications and I/Os are generated by the compiler as an XML file.

Model Driven Development extension translates SysML diagrams [40] into templates of programs or communication tasks [29]. Another extension supports unit testing of POUs (Program Organization Units) [27]. All CPDev modules have XML interfaces to enable agile and round-trip software development [28].

Extended runtime environment besides the VM for control functions involves HMI runtime for the interface. The VM and HMI runtime are executed in parallel exchanging data by means of global variables.

4. Intermediate VMASM and Compilation

4.1. Overview of VMASM

As an IEC 61131-3-oriented language, Virtual Machine Assembler accepts all elementary data types except `WSTRING`. Number of types required by particular application may be restricted by parametrizing the compiler. VMASM instructions consist of functions and procedures. Functions are the same as in the standard, so ADD, OR, EQ, CONCAT, EXPT, etc. They admit up to 16 operands where the first one denotes the result. Procedures shown in Table 1 (examples) are typical for machine languages, so they control program flow, call subprograms, copy memory, etc.

Tab. 1. Procedures of the VMASM language

Mnemonic	Meaning
JMP	Unconditional jump
JZ	Conditional jump
JR	Unconditional relative jump
CALB	Subroutine call
RETURN	Return from subroutine
MCD	Initialize data
MEMCP	Copy memory block
FPAT	Fill memory block

Syntax of VMASM instructions is the following:

```
[:label] instruction [operand1]
[,operand2][,operand3]...
```

Label is optional, instruction specifies number of operands, i.e. variables, labels or constants. Since in case of functions `operand1` denotes a variable (result), so the syntax expresses memory-to-memory type of operation. Question mark `?` at the beginning indicates label or in `?LR?` prefix a temporary variable created by the compiler. Dot operator (`.`) selects components of arrays or structures.

Note that in IL language, often used as intermediate (Sec. 2), result of a command is kept in Current Result (CR) register equivalent to accumulator. Later the CR is copied into a variable. By placing the result in the `operand1`, VMASM combines these two steps into one. The notion of accumulator does not exist in VMASM.

4.2. Parametrized Compiler

To ensure portability of implementations one has to take primarily into account sizes of addresses required by CPUs, so typically 16 or 32 bits. Also some data types may be not needed. To provide such features the CPDev compiler is parametrized by means of an XML Library Configuration File (LCF) involving definitions of hardware resources, data types, instructions and conversions. Given the parametrized LCFs dedicated compilers and VMs for particular platforms can be created from one general compiler and generic VM.

```
<HARDWARE>
  <AddressSize>2</AddressSize>
  <MaxCodeAddress>0xFFFF</MaxCodeAddress>
  <MaxDataAddress>0xFFFF</MaxDataAddress>
</HARDWARE>
```

```
<TYPES>
  <type name="UINT" implement="alias">
    <alias name="WORD"/>
  </type>
  <deny-type name="LREAL" />
</TYPES>
```

Fig. 2. Specification of hardware and data types

Sample specification of `<Hardware>` resources in

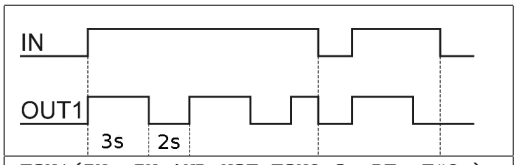
the LCF file is shown in Fig. 2 (upper part) with AddressSize and MaxAddresses of code and data memories available for VM. 2 as AddressSize means 2 bytes needed for 16-bit addressing of up to 64 kB memory (0xFFFF). 4 represents 32-bit addressing of 4 GB. Note that the AddressSize defines addressing for VM only, so 16-bit VM may be executed by a 32 or 64-bit CPU.

<TYPES> part of the LCF file (Fig. 2) defines data types available in particular implementation. By using deny-type one can remove not needed types, for instance LREAL. The deny-type option enables configuration of the environment for so-called *made-to-measure PLCs*, suitable particularly for IoT and Industry 4.0 applications.

Given the LCF file, scanner and parser of the compiler translate the source program into a file with VMASM mnemonics. By using another set of keywords the ST scanner also processes IL, where ad hoc temporary variables replace CR register. The parser applies top-down syntax-directed translation [8]. Basic components of the compiler are defined as classes in C# language. Internal data are kept in lists.

4.3. Compilation Example

ST program for switching OUT on-and-off in 3+2 seconds cycle while IN is a active, and VMASM translation of the first TON1 call are shown in Fig. 3. A warning light may be triggered in this way.




```
TON1(IN:=IN AND NOT TON2.Q, PT:=T#3s);
TON2(IN:=IN AND TON1.Q, PT:=T#2s);
OUT := IN AND NOT TON1.Q;
```

```
  JZ IN, :?WARN?AN10
  NOT ?LR?AN11, TON2.Q
  JZ ?LR?AN11, :?WARN?AN10
  MCD TON1.IN, #01, #01
  JMP :?WARN?EA14
:?WARN?AN10
  MCD TON1.IN, #01, #00
:?WARN?EA14
  MCD TON1.PT, #04, #B80B0000
  CALB TON1, :?IEC_61131.TON?CODE
```

Fig. 3. ST code and part of VMASM translation

The compiler translates Boolean expressions involving infix operations like IN AND TON2.Q by means of value testing and jumps. Hence the VMASM translation begins with jump JZ to :?WARN?AN10 label if IN is zero. If not, NOT of TON2.Q is evaluated in ?LR?AN11 temporary variable, followed by another JZ to the same label if the variable is zero. If not, MCD instruction sets 1 byte (first #01) at TON1.IN to 1 (second #01), followed by unconditional JMP to :?WARN?EA14. At the :?WARN?AN10 label another MCD sets TON1.IN to 0 (#00). The third MCD at :?WARN?EA14 initializes TON1.PT (4 bytes, TIME) with #B80B0000, i.e. 2000 ms (little endian form). Final CALB calls code of TON1 block from IEC_61131 library. Value testing and jumps could be avoided by writing the Boolean expression in function form as AND(IN,TON2.Q)).

Translation of TON2 call is similar. OUT statement is translated in the same way as the expression for TON1.IN. Optimization of the mnemonic translation to minimize the number of temporary variables precedes generation of binary code.

5. Binary Code and Virtual Machine

5.1. Assembling the Mnemonics

To generate the binary code, VMASM instructions are defined in the LCF file by means of digital identifiers vmcode composed of group ig and type it subidentifiers as shown in Fig. 4a. In case of functions ig indicates name, so ADD, MUL, AND, EQ, etc., whereas it refers to data type, BOOL, INT, TIME and so on. In this way type-specific functions such as ADD:INT, AND:BOOL and others may be expressed in digital form. Fig. 4b (upper part) shows definition of AND:BOOL where ig=08 denotes AND and it=*0 indicates BOOL (0) and varying number of inputs (*).

All VMASM procedures belong to one group ig=1C with it choosing specific procedure. Definition of MCD that initializes data memory beginning from DST (destination) relative label (:rdlab) with imm1 bytes (immediate value) from the source imm2 of varying type (*) is also in Fig. 4b (lower part).

ig	it
group identifier	*type identifier*

(a)

```
<function name="AND" vmcode="08*0" return="BOOL">
 <args>
  <arg no="*" name="arg*" type="BOOL"/>
 </args>
 <comment>Binary AND of BOOL operands</comment>
</function>
```

```
<sysproc name="MCD" vmcode="1C15">
 <args>
  <arg no="0" name="DST" type=":rdlabel"/>
  <arg no="1" name="imm1" type=":imm.BYTE"/>
  <arg no="2" name="imm2" type=":imm.*"/>
 </args>
 <comment>Initializes data at arg0 address with area
 size arg1 and source pattern arg2.</comment>
</sysproc>
```

(b)

Fig. 4. a) Structure of vmcode, b) specification of function and procedure

Having the file with VMASM mnemonics all program modules are consolidated and assembled into binary code by replacing:

- instruction mnemonics by digital identifiers vmcode (ig and it)
- names of variables by addresses in data memory

- labels by addresses in code memory.

Binary code assembled from the first four VMASM instructions in Fig. 3 looks as follows:

```
1C02 0000 B900
0510 0200 2100
1C02 0200 B900
1C15 0800 0101
```

The `vmcodes` of `JZ`, `NOT` and `MCD` instructions are `1C02`, `0510`, `1C15`, respectively. `IN` is allocated to address `0` in data memory, `TON2.Q` to `21` (hexadecimal), temporary `?LR?AN11` to `2`, and `TON1.IN` to `8`. The label `:?WARN?AN15` indicates `B9` in the code memory.

5.2. Virtual Machine

General architecture of CPDev virtual machine involving code and data memories, instruction processing module, and target platform interface is shown in Fig. 5. The module fetches instructions from code memory, executes them acquiring operands from data or code memories (variables or constants) and in case of functions stores the results in data memory. Procedures change internal state of VM. Target platform interface in the generic VM consists of specifications of low-level functions filled in by appropriate code while porting. VM components are implemented as 16 or 32-bit according to `AddressSize`.

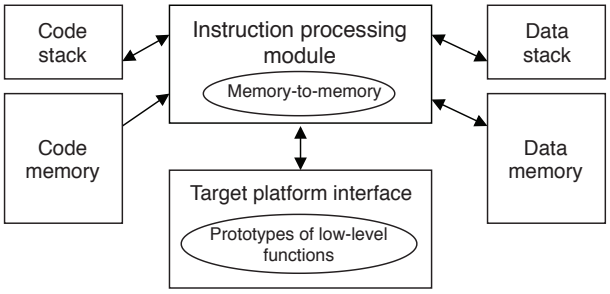


Fig. 5. General architecture of the virtual machine

Generic part of VM is written in standard C, so suits any general purpose CPU. Note that the VMs mentioned in Section 2 are mostly written in C. The VM operates as an interpreter executing successive lines of code. Overloaded functions such as `AND`, `MUL`, `GT`, etc. accept all meaningful data types. Variable number of operands is handled by a `for` loop. A procedure indicated by `ig=1C` is selected by `switch(it)` command.

Specifications of low-level functions included into the generic VM are expressed by function prototypes (empty), independent of CPU and hardware solution. Initial set of prototypes has been presented in [54] based on experience with multifunction instruments [7] and industrial controllers. Some of the prototypes are listed below:

- `VMP_LoadConfiguration` – loads task parameters (cycle, number of POUs, etc.), binary code and allocates memory for data
- `VMP_PreRunConfiguration` – initializes hardware and stores the initial state (time, parameters)
- `VMP_PreCycle` – updates program variables from external inputs before calculations, stores initial state of system clock
- `VMP_PostCycle` – updates external outputs with results of calculations; if time is left triggers tests or other activities
- `VMP_CurrentTime` – returns current value of system clock to determine `TIME` variables.

In case of cycle time overrun, the `VMP_PostCycle` sets a flag in the VM status. The VM checks the flag and takes appropriate steps (failure LED, safe outputs, etc.).

By assumption, the code that fills the prototypes is written in C/C++ by implementation engineer and consolidated with the generic VM. Some other hardware functions may be added.

As indicated in the introduction the use of VM lengthens the execution cycle as compared to direct translation to machine code or to C/C++. To evaluate the overhead, execution times of standard function blocks such as flip-flops, edge detectors, counters and timers have been measured, at first implemented in ST and then in C by means of so-called native blocks (Sec. 7.1). By average, the VM solution has turned out about four times slower than translation to C/C++ [16], [53]. So the use of native blocks reduces the overhead considerably.

5.3. Runtime for Complex Software

The generic VM described above executes single project involving one task. However, deployment of a few VMs in a multi-core FPGA has been tested already [18], leading the way to development of multi-project runtime needed for a supervisory computer or software processing station. To make implementation relatively simple it has been assumed that operating system of the computer will take care of scheduling or multitasking of the VMs.

This concept has been recently transferred into runtime for industrial PC with Windows Embedded operating system. The runtime is called WinController and can execute complex software composed of projects run by corresponding VMs with different or the same cycles. The VMs exchange data through global variables. Field devices, namely PLCs, PACs, I/O cards etc. are interfaced by means of special I/O and Data adapters. The WinController is implemented as a service in the Windows service system functionality. It runs nonstop in the background and starts automatically when the computer is switched on.

The service is configured by WinController Manager with relevant part of main window shown in Fig. 6. Besides Service status, I/O and Data adapters, the window includes two VMs running AUTOPILOT and HMI projects. Data exchange is configured by I/O mappings.

6. Graphical Translators and HMI Tool

6.1. LD and FBD Translators to ST

Although LD and FBD diagrams look different, CPDev editors are similar partly due to the same A*

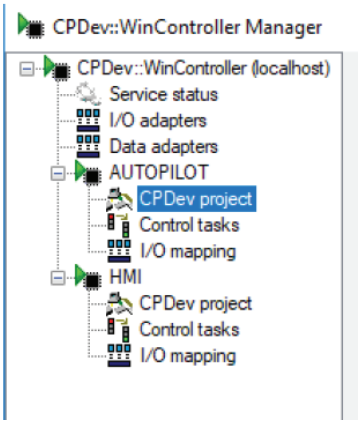
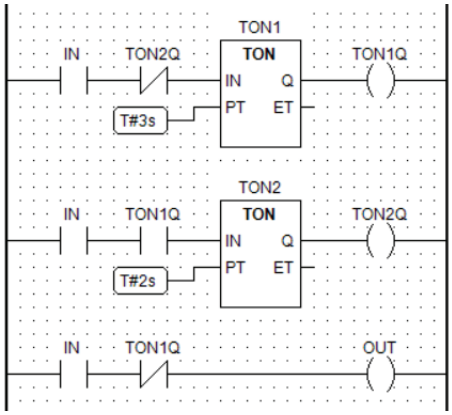


Fig. 6. Configuration window of WinController Manager

pathfinding algorithm used for automatic connections [19]. Translation replaces the connections by auxiliary local variables to display 'live" diagrams in the online mode. Actually the vertices of corresponding AOV graph represent diagram elements whereas virtual nodes denote connections. Each line of ST program reflects the node.

Three-rung LD version of the running example followed by ST translation of the first rung is shown in Fig. 7. The auxiliary variables beginning with `out_contact_` are declared automatically in `VAR ... END_VAR` section. Besides the connections the variables represent branch points and outputs of function blocks or functions.

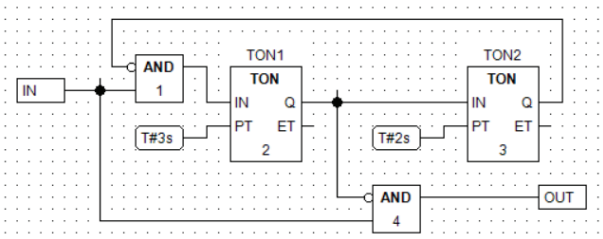



```
out_contact_IN_80_80 := IN;
out_contact_TON2Q_140_80 := out_contact_IN_80_80 AND NOT TON2Q;
TON1(IN:=out_contact_TON2Q_140_80,PT:=T#3s,Q=>out_TON1_Q);
TON1Q := out_TON1_Q;
```

Fig. 7. FBD diagram and ST translation

FBD diagram and corresponding translation are shown in Fig. 8. Here the prefix `out_` indicates auxiliary variables. Except for such variables the translation does not differ from the ST original in Fig. 3.

To make the automatically created connections looking more or less like drawn "by hand" the original A* algorithm has been extended to penalize path crossings and direction changes [13]. Note also that after each correction of the diagram the algorithm calculates all paths anew, so they may be a little different than before.




```
out_and1 := AND(NOT out_TON2_Q, IN);
TON1(IN:=out_and1,PT:=T#3s,Q=>out_TON1_Q);
TON2(IN:=out_TON1_Q,PT:=T#2s,Q=>out_TON2_Q);
out_and2 := AND(NOT out_TON1_Q, IN);
OUT := out_and2;
```

Fig. 8. LD diagram and ST translation

6.2. SFC Translation

SFC diagram for the running example is shown in Fig. 9. Actions set or reset the `OUT` output. The diagrams composed of steps, transitions, jumps, sequence selections and simultaneous sequences are kept in memory as trees. The first three elements create the last two. Actions bound to steps are defined by qualifiers such as `N` Normal/Non-stored, `S` Set, `R` Reset, `L` time Limited, `D` Delayed and other [22].

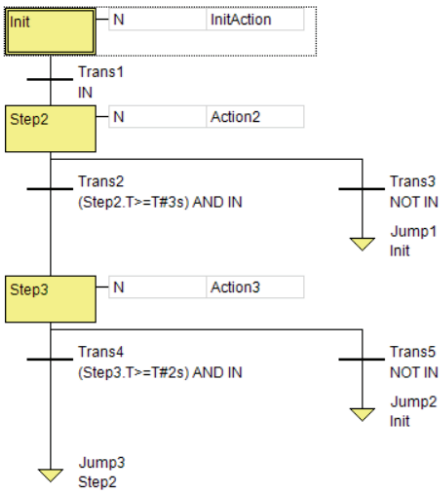


Fig. 9. SFC diagram for the running example

Translation of SFC to ST is supported by two system function blocks, `SFC_STEP` and `SFC_ACTION_CONTROL`, connected as in Fig. 10. The first one determines step activity according to the `next_step_flag`. The outputs `X`, `T` indicate activity and time elapsed since activation (see Fig. 9). By means of the `A` output the second block triggers action (code) bound to the step according to given qualifier. This is provided by connecting the output `X` of the first block to the input of the second block indicated by the qualifier (`N`, `S`, `R`, `L`, `D`, ...).

ST translation of SFC diagram begins with declarations of the instances of `SFC_STEP` and `SFC_ACTION_CONTROL`, and `next_step_flags`. `PROGRAM` section consists of three parts:

- activation of step blocks by `next_step_flags`
- execution of actions bound to active steps

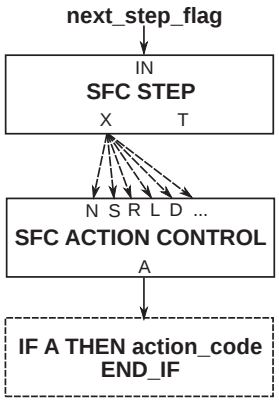


Fig. 10. Execution of step-action pair

- setting `next_step_flags` by evaluation of transition conditions.

The first two parts correspond to Fig. 10 while the last one creates the tree structure according to SFC diagram. Simultaneous sequence may set/reset a few `next_step_flags` by a single preceding transition.

As explained in [4], SFC diagrams may exhibit unreachable and unsafe behaviour. Unreachability is partially restricted by the editor but the designer's task is to avoid unsafe diagrams.

6.3. Human Machine Interface

Small control systems based on PLCs and PACs often include graphical operator panels. The panels may be integrated with controllers, as in case of Horner, Unitronix, Beckhoff and others.

HMI software for such panels consists of two parts related to visualization and behaviour. Visualization involves a set of displays built from graphic objects selected from libraries. Behaviour is determined by HMI programs that select displays and specify appearance of the objects. To do so, relevant variables are exchanged between the controller and panel. The variables are bound to attributes of the objects during configuration. If control and HMI programs belong to the same project they access common global variables to exchange data.

Features outlined above are provided in the environment by CPVis tool involving a dedicated graphic editor for defining displays, adding objects, moving them, etc. Attributes of the objects are determined by global variables or constants. XML data of the project converted into binary code are used in the runtime environment where VM and CPVis runtime jointly provide control and HMI functions. Example of display for power management at a ship is shown in Fig. 11.

By splitting the HMI software into device-dependent and device-independent parts proposed in [30], visualizations created by CPVis can be displayed by different devices, i.e. TFT or LCD panels, monitors or tablets. The device-dependent parts are specified by common prototypes of low-level drawing functions called *drawing primitives*. Codes of graphic objects call the prototypes by name only, so they are device-independent, as depicted in Fig. 12. Two drawing primitives concerning a rectangle are shown

Fig. 11. TFT display for power management at a ship [43]

below:

- `DrawRectangle`: position, size, border color and width
- `FillRectangle`: position, size, fill color.

By assigning a variable to `FillRectangle`'s size the two primitives may create a bargraph. `DrawArc`, `FillPie`, `FillTriangle`, `ProcessValue` are examples of other primitives.

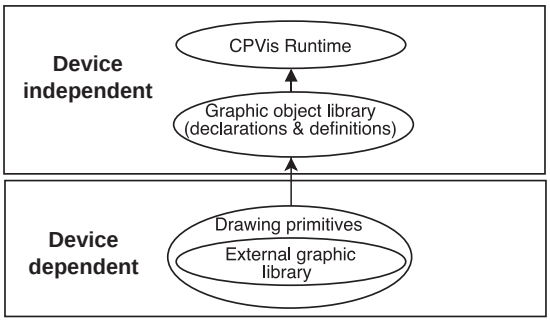


Fig. 12. Structure of the HMI tool

Implementation of CPVis for particular display device requires filling the drawing primitives with appropriate code using the mechanisms and library specific for the device, for instance RamTex, GLCD or GDI+. Custom graphic objects are created by writing the code in C with calls of drawing primitives. Such objects can be displayed by another device provided that the drawing primitives are prepared for it. Notice that the concept of drawing primitives is similar to the prototype functions specified for the target platform interface (Sec. 5.2).

7. Communication and Implementations

7.1. Structure of Communication Data

By the original assumptions, the implementation engineers themselves develop configuration tools while porting the environment to particular platforms (along with target platform interfaces). To support this the compiler-generated XML file (DCP extension), besides binary code and VMASM mnemonics (for debugging), also contains relevant data on global variables. The data are kept in `VAR` entities, such as those

in Fig. 13 for IN and OUT variables from the running example. The successive entries denote:

- LName – local name inside current variable scope
- PName – physical name including project name scope
- Addr – hexadecimal address (16-bit version in Fig. 13)
- AddrType – address type; :gdlabel means global data label
- Size – variable size in bytes
- Type – variable type
- PType – physical type; $Default scope involves system elements
- VarFlags – flags related to variable.

Variables imported from libraries have LIBRARY.VARIABLE as physical name to avoid conflicts. Besides elementary types, AddrTypes include data label relative to current POU (:rdlabel), global or relative labels to code (:gclabel, :rclabel), array and structures indicated by ?ARRAY?NUM and ?STRUCT?NUM (unique NUM). Flags involve: 1 – retain, 4 – constant, 8 – variable with address defined by AT, 0x4000 – global, 0x800000 – data channel. Other flags may be defined.

```
<VAR LName="IN" PName="RevSwitchST.IN" Addr="0000"
AdrType="gdlabel" Size="1" Type="BOOL"
PType="$DEFAULT.BOOL" VarFlags="00004000" />
<VAR LName="OUT" PName="RevSwitchST.OUT" Addr="0001"
AdrType="gdlabel" Size="1" Type="BOOL"
PType="$DEFAULT.BOOL" VarFlags="00004000" />
```

Fig. 13. <VAR> entities in the DCP file

Depending on software solution, local or physical names and addresses may be used by configuration tools to bind variables to interfaces. The original paper [7] presented CPCon tool to interface remote I/Os by means of physical data. Local data sufficed to configure SCADA InTouch for supervisory PC.

An OPC server can also be designed given XML DCP file. In fact, simple server adapted from publicly available LightOPC [33] is built-in into the CPDev softcontroller.

Besides the DCP-based configuration tool a few variables can be interfaced to dedicated external ports by using native block concept (as in *Java Native Interface* or *.NET Platform Invoke*). It was originally described in [54] and applied for GPS module with NMEA serial communication. Native blocks are function blocks written in C/C++ but, unlike typical function blocks, they are deployed on the target platform as components of the VM, similarly as the target platform interface. To some extent they resemble hardware blocks from multifunction instruments. Software algorithms implemented as C/C++ native blocks are executed much faster than blocks written in IEC 61131-3 languages (Sec. 5.2).

7.2. Characteristic Implementations

As may be concluded from above, the implementation engineer can port CPDev environment to particular platform after development of:

- target platform interface
- communication and I/O configuration tool
- deployment module to upload the binary code and configuration data into the controller.

Upgrading existing devices such as dedicated controllers, distributed I/O units or data loggers with IEC 61131-3 capabilities has been the main purpose of the implementations so far. Therefore the engineers could use major portions of low-level software while porting. Deployment has involved Ymodem for serial, ftp or tftp for Ethernet and GSM/GPRS for wireless.

Power management, propulsion control, heading control and other subsystems belonging to MegaGuard ship automation and navigation system from Praxis [43] are most significant implementations of CPDev. Each subsystem involves a control processor, I/O unit (units) and TFT touch-panel (as in Fig. 11), all of them equipped with ARM CPUs. CAN is applied for communication with I/O and Ethernet (redundant) for TFT. The processor for controlling a setup composed of diesel engine, electric generator and circuit breaker is shown in Fig. 14. Praxis software is written in ST. TFT displays consist of components from user-defined libraries. NMEA serial/Ethernet protocol is used by marine electronic devices.

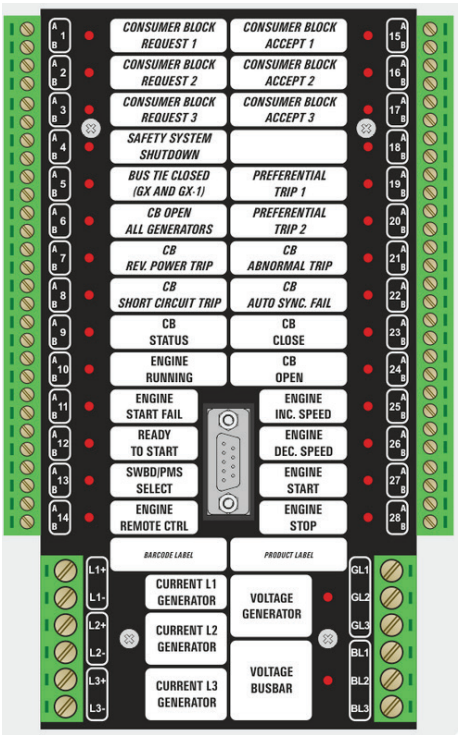


Fig. 14. Power management processor [43]

Substation automation and medium voltage Grid control is provided by Remote Telecontrol Units (RTUs) from iGrid T&D [24] shown in Fig. 15. RTUs consist of ARM processor, I/O board and communication interfaces for a number protocols including

IEC 60870 and 61850 dedicated for power systems. Documentation generation with complicated FBD and LD diagrams is important.

Fig. 15. Remote Telecontrol Unit [24]

Mini-DCS system from LUMEL described in [7] was the first domestic implementation of CPDev (also found in Philippines [25]). StTr-PLC controller from NiT [37] for control of communal facilities distributed over large area, such as water supplies, heating stations or sewage pumps, is another one. StTr-PLCs are monitored by GSM/GPRS integrated communication platform also used for mobile applications. Recently developed PLC1 controller from BartCom [3] with local or distributed I/Os (Modbus, Profibus) is dedicated for intelligent homes and general automation.

8. Conclusions

Technical aspects of current features of CPDev multi-platform engineering environment have been presented. The runtime is based on virtual machine executing intermediate VMASM code into which ST and IL programs are compiled. Programs written in graphical languages are translated into ST. Parametrization of the compiler and VM for commonly used 16- and 32-bit addressing and for eventual restriction of data types is provided by the Library Configuration File. Implementations on diverse hardware platforms are supported by prototypes of low-level functions introduced into the generic VM. HMI projects split into device-independent and device-dependent parts may be displayed by different devices. General purpose XML file with compiler-generated data on global variables support configuration of communications and I/Os. Native blocks may considerably decrease the VM overhead. Implementations have shown that equipment manufacturers themselves are able to port the environment to relevant devices.

Object-oriented programming and moving the multi-project runtime into multi-core processors, with each core executing a single project, are currently under investigation.

AUTHORS

Dariusz Rzońca* – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: drzonca@kia.prz.edu.pl.

Jan Sadolewski – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: js@kia.prz.edu.pl.

Andrzej Stec – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: astec@kia.prz.edu.pl.

Zbigniew Świder – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: swiderzb@kia.prz.edu.pl.

Bartosz Trybus – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: btrybus@kia.prz.edu.pl.

Leszek Trybus – Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland, e-mail: ltrybus@kia.prz.edu.pl.

*Corresponding author

ACKNOWLEDGEMENTS

Help from Marcin Jamro a few years ago is acknowledged.

This project is financed by the Minister of Science and Higher Education of the Republic of Poland within the ”Regional Initiative of Excellence” program for years 2019 – 2022. Project number 027/RID/2018/19, total amount granted 11 999 900 PLN.

REFERENCES

[1] 3S-Smart Software Solutions GmbH. “CODESYS”. www.codesys.com. Accessed on: 2020-02-28.

[2] Axel S.r.l. “Logiclab”. www.axelsoftware.it/en/. Accessed on: 2020-02-28.

[3] Bart-Com. www.bart-com.pl. Accessed on: 2020-02-28.

[4] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell. “A Unifying Semantics for Sequential Function Charts”. In: H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, eds., *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, Lecture Notes in Computer Science, 400–418. Springer, Berlin, Heidelberg, 2004, 10.1007/978-3-540-27863-4_22.

[5] S. Cavalieri, G. Puglisi, M. S. Scroppo, and L. Galvagno, “Moving IEC 61131-3 applications to a computing framework based on CLR

Virtual Machine". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, 1–8, 10.1109/ETFA.2016.7733632.

[6] M. Chmiel and E. Hrynkiewicz, "Concurrent operation of processors in the bit-byte CPU of a PLC", *Control and Cybernetics*, vol. 39, no. 2, 2010, 559–579.

[7] J. Cisek, W. Mikluszka, Z. Swider, and L. Trybus, "A Low-Cost DCS with Multifunction Instruments and CAN Bus1", *IFAC Proceedings Volumes*, vol. 34, no. 29, 2001, 64–69, 10.1016/S1474-6670(17)32794-5.

[8] K. D. Cooper and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann: Boston, 2012, 10.1016/C2009-0-27982-7.

[9] COPA-DATA France. "STRATON–PLC". www.straton-plc.com. Accessed on: 2020-02-28.

[10] W. Dai, V. Vyatkin, J. H. Christensen, and V. N. Dubinin, "Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability", *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, 2015, 771–781, 10.1109/TII.2015.2423495.

[11] G. De Tommasi and A. Pironti, "An educational open-source tool for the design of IEC 61131-3 compliant automation software". In: *Automation and Motion 2008 International Symposium on Power Electronics, Electrical Drives*, 2008, 486–491, 10.1109/SPEEDHAM.2008.4581144.

[12] G. Fen and W. Ning, "A Transformation Algorithm of Ladder Diagram into Instruction List Based on AOV Digraph and Binary Tree". In: *TENCON 2006 - 2006 IEEE Region 10 Conference*, 2006, 1–4, 10.1109/TENCON.2006.343937.

[13] E. Ferreira, R. Paulo, C. D. Da, and P. Henriques, "Integration of the ST language in a model-based engineering environment for control systems: An approach for compiler implementation", *Computer Science and Information Systems*, vol. 5, no. 2, 2008, 87–101, 10.2298/CSIS0802087F.

[14] J.-Y. Fiset, *Human-machine interface design for process control*, Instrumentation, Systems, and Automation Society: Research Triangle Park, NC, 2009.

[15] P. Gaj, J. Jasperneite, and M. Felser, "Computer Communication Within Industrial Distributed Environment—a Survey", *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, 2013, 182–189, 10.1109/TII.2012.2209668.

[16] GEB Automation. "GEB IDE". www.gebautomation.com. Accessed on: 2020-02-28.

[17] P. Gsellmann, M. Melik-Merkumians, and G. Schitter, "Comparison of Code Measures of IEC 61131–3 and 61499 Standards for Typical Automation Applications". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, 1047–1050, 10.1109/ETFA.2018.8502464.

[18] Z. Hajduk, B. Trybus, and J. Sadolewski, "Architecture of FPGA Embedded Multiprocessor Programmable Controller", *IEEE Transactions on Industrial Electronics*, vol. 62, no. 5, 2015, 2952–2961, 10.1109/TIE.2014.2362888.

[19] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968, 100–107, 10.1109/TSSC.1968.300136.

[20] L. Huang, W. Liu, and Z. Liu, "Algorithm of transformation from PLC ladder diagram to structured text". In: *2009 9th International Conference on Electronic Measurement Instruments*, 2009, 4–778–4–782, 10.1109/ICEMI.2009.5274701.

[21] ICS Triplex. "ISaGRAF". www.isagraf.com. Accessed on: 2020-02-28.

[22] IEC. "IEC 61131-3 – Programmable controllers – Part 3: Programming languages", 2013.

[23] IEC. "IEC 61499 – Function Blocks", 2012.

[24] iGrid T&D. www.igrid-td.com. Accessed on: 2020-02-28.

[25] Instrument Science Systems, Inc. www.issi.com.ph/. Accessed on: 2020-02-28.

[26] H. I. Inzunza Villagómez, B. Pérez Arce, S. I. Hernández Ruiz, and J. A. López Corella, "Design and implementation of a development environment on ladder diagram (HT-PLC) for Arduino with Ethernet connection". In: *2018 IEEE International Conference on Automation/XXIII Congress of the Chilean Association of Automatic Control (ICA-ACCA)*, 2018, 1–6, 10.1109/ICA-ACCA.2018.8609850.

[27] M. Jamro, "POU-Oriented Unit Testing of IEC 61131-3 Control Software", *IEEE Transactions on Industrial Informatics*, vol. 11, no. 5, 2015, 1119–1129, 10.1109/TII.2015.2469257.

[28] M. Jamro and D. Rzonca, "Agile and hierarchical round-trip engineering of IEC 61131-3 control software", *Computers in Industry*, vol. 96, 2018, 1–9, 10.1016/j.compind.2018.01.004.

[29] M. Jamro, D. Rzonca, and W. Rząsa, "Testing communication tasks in distributed control systems with SysML and Timed Colored Petri Nets model", *Computers in Industry*, vol. 71, 2015, 77–87, 10.1016/j.compind.2015.03.007.

[30] M. Jamro and B. Trybus, "IEC 61131-3 programmable human machine interfaces for control devices". In: *2013 6th International Conference on Human System Interactions (HSI)*, 2013, 48–55, 10.1109/HSI.2013.6577801.

[31] H. S. Kim, J. Y. Lee, and W. H. Kwon, "A compiler design for IEC 1131-3 standard languages of programmable logic controllers". In:

SICE '99. Proceedings of the 38th SICE Annual Conference. International Session Papers (IEEE Cat. No.99TH8456), 1999, 1155–1160, 10.1109/SICE.1999.788715.

[32] Y. Lee, J. Jeong, and Y. Son, "Design and implementation of the secure compiler and virtual machine for developing secure IoT services", *Future Generation Computer Systems*, vol. 76, 2017, 350–357, 10.1016/j.future.2016.03.014.

[33] Light OPC Server. www.ipi.ac.ru. Accessed on: 2020-02-28.

[34] D. Massonie, C. Hacker, and T. Sowa, "Modeling Graphical and Speech User Interfaces with Widgets and Spidgets". In: *Speech Communication; 11. ITG Symposium*, 2014, 1–4.

[35] A. Milik, "Multiple-Core PLC CPU Implementation and Programming", *Journal of Circuits, Systems and Computers*, vol. 27, no. 10, 2018, 1850162, 10.1142/S0218126618501621.

[36] A. Milik and E. Hrynkiewicz, "Distributed PLC Based on Multicore CPUs - Architecture and Programming", *IFAC-PapersOnLine*, vol. 49, no. 25, 2016, 1–7, 10.1016/j.ifacol.2016.12.001.

[37] Nauka i Technika. www.nit.pl. Accessed on: 2020-02-28.

[38] nxtControl GmbH. www.nxtcontrol.com/. Accessed on: 2020-02-28.

[39] M. Okabe, "Development of processor directly executing IEC 61131-3 language". In: *2008 SICE Annual Conference*, 2008, 2215–2218, 10.1109/SICE.2008.4655032.

[40] OMG. "OMG Systems Modeling Language, V1.3", 2012.

[41] L. B. Palma, V. Brito, J. Rosas, and P. Gil, "WEB PLC simulator for ST programming". In: *2017 4th Experiment@International Conference (exp.at'17)*, 2017, 303–308, 10.1109/EXPAT.2017.7984410.

[42] C. G. Penteado, E. D. Moreno, and F. D. Pereira, "A microcontroller multicore in FPGAs: detailed architecture and case studies of embedded critical applications", *International Journal of Grid and Utility Computing*, vol. 8, no. 3, 2017, 169, 10.1504/IJGUC.2017.087815.

[43] Praxis Automation Technology B.V. www.praxis-automation.nl. Accessed on: 2020-02-28.

[44] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic Configuration of GCC Using Irace". In: E. Lutton, P. Legrand, P. Parrend, N. Monmarché, and M. Schoenauer, eds., *Artificial Evolution*, Cham, 2018, 202–216, 10.1007/978-3-319-78133-4_15.

[45] J. Quiroga and F. Ortin, "SSA Transformations to Facilitate Type Inference in Dynamically Typed Code", *The Computer Journal*, vol. 60, no. 9, 2017, 1300–1315, 10.1093/comjnl/bxw108.

[46] J. Richter, *CLR via C#*, Microsoft Press: Redmond, Washington, 2012.

[47] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Mini-DCS system programming in IEC 61131-3 structured text", *Journal of Automation Mobile Robotics and Intelligent Systems*, vol. 2, no. 3, 2008, 48–54.

[48] S. Shin, M. Kwon, and S. Rho, "Whimori CDK: A Control Program Development Kit". In: *Engineering and Information 2009 International Conference on Computing*, 2009, 115–118, 10.1109/ICC.2009.33.

[49] H. Simon and S. Kowalewski, "Static analysis of Sequential Function Charts using abstract interpretation". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, 1–4, 10.1109/ETFA.2016.7733648.

[50] T. Skripcak, P. Tanuska, U. Konrad, and N. Schmeisser, "Toward Nonconventional Human–Machine Interfaces for Supervisory Plant Process Monitoring", *IEEE Transactions on Human-Machine Systems*, vol. 43, no. 5, 2013, 437–450, 10.1109/THMS.2013.2279006.

[51] VDI/VDE 3699 Process control using display screens, 2015.

[52] K. Thramboulidis, "A cyber–physical system-based approach for industrial automation systems", *Computers in Industry*, vol. 72, 2015, 92–102, 10.1016/j.compind.2015.04.006.

[53] E. Tisserant, L. Bessard, and M. de Sousa, "An Open Source IEC 61131-3 Integrated Development Environment". In: *2007 5th IEEE International Conference on Industrial Informatics*, vol. 1, 2007, 183–187, 10.1109/INDIN.2007.4384753.

[54] B. Trybus, "Development and Implementation of IEC 61131-3 Virtual Machine", *Theoretical and Applied Informatics*, vol. 23, no. 1, 2011, 10.2478/v10179-011-0002-z.

[55] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill: New York, 1997.

[56] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach", *Mechatronics*, vol. 24, no. 7, 2014, 883–897, 10.1016/j.mechatronics.2014.05.003.

[57] M. Zhang, Y. Lu, and T. Xia, "The Design and Implementation of Virtual Machine System in Embedded SoftPLC System". In: *2013 International Conference on Computer Sciences and Applications*, 2013, 775–778, 10.1109/CSA.2013.185.

[58] C. Zhou and H. Chen, "Development of a PLC Virtual Machine Orienting IEC 61131-3 Standard". In: *2009 International Conference on Measuring Technology and Mechatronics Automation*, vol. 3, 2009, 374–379, 10.1109/ICMTMA.2009.422.

[59] R. Zurawski, ed., *Industrial Communication Technology Handbook*, Industrial information

technology series, CRC Press, Taylor & Francis Group: Boca Raton London New York, 2015.