My personal comments on IEC61131-3 standard programming languages

Wojciech GOMOLKA
FESTO France

TABLE OF CONTENTS

1. Introduction: 2
2. LD: Ladder Diagram (Contact language) 3
3. FBD: Function Block Diagram 6
4. IL : Instruction List 8
5. ST: Structured Text 10
6. SFC: Sequential Function Chart 12
Conclusions: choose a language suited to your project 14
Appendix 1: Summary of advice found on the web: 16
Appendix 2: CFC: Continuous Function Chart 17

1. Introduction:

Generally, programmable logic controller (PLC) providers offer their own, specific programming platforms.
Yet, over the last decade, the main European, Asian and American PLC providers have started to offer new platforms based on the IEC61131-3 standard.

The IEC61131-3 standard aims to standardise controller programming for specific elements such as Data Types, Programming Languages, Functions and basic Function Blocks. For example: counters, timers, triggers, etc.
The standard also suggests a standardization of the implementation of programmable modules POU's (such as Programs, Functions, Function Blocks) in order to encourage the creation of reusable and transferrable (portable) applications for different equipment.

The IEC 61131-3 standard defines five programming languages, their syntax and semantics.
These languages are:

- **LD**: Ladder Diagram (contact language)
- **IL**: Instruction List
- **FBD**: Function Block Diagram
- **ST**: Structured Text
- **SFC**: Sequential Function Chart

But with the increasing complexity of application and their implementations, it would be nice to become familiar with the characteristics of each language:

- What are the advantages and disadvantages of each one?
- When one language could be used (or preferred) instead of another?

This note briefly discusses and compares each of these 5 main PLC programming languages.

Remark 1
To illustrate, take for example the calculation of a product's net weight, for which the gross amount is indicated by a toggle in BCD format. The result is also converted into BCD displayed by an indicator.

NetWeight: = INT_TO_BCD(BCD_TO_INT(TotalWeight) – iTareWeight):

The TotalWeight and NetWeight variables are coded in BCD and the iTareWeight is in integer type (INT). The StartCalculation variable is a Boolean variable (BOOL).

Remark 2 (personal)
This is a rather lengthy article and the subject is well-known.
Thus, reading it may cause serious damage such as loss of time, back pain, pain in your fingers; wear and tear of the mouse and keyboard; etc.
Please ensure that you are determined and motivated enough to continue reading or, go directly to the conclusion. Otherwise, select a different, shorter article.

2. LD: Ladder Diagram (Contact language)

This programming language was invented in the United States of America several decades ago. It was created in order to program the first PLC controllers that replaced hard-wired relay control systems (relay cabinets).

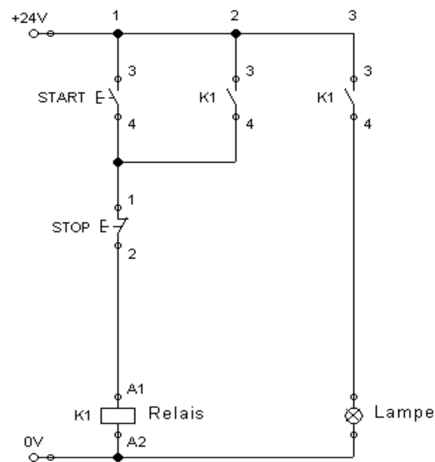

Fig. 1 Example of a relay network: Control circuit for a lamp

The program written in this language is visually like an electric wiring diagram (network) that includes contacts, coils, and connections.

The Ladder language consists of 3 basic elements:

- Contacts (or inputs) that can read the value of Boolean variables
- Coils (or outputs) that allow to write the value of Boolean variables
- Function Blocks or Functions that allow to perform more complex operations using numerical variables

Each element corresponds to a variable; either Boolean (e.g. contact, coil) or numerical (e.g. inputs/outputs of Function Block).

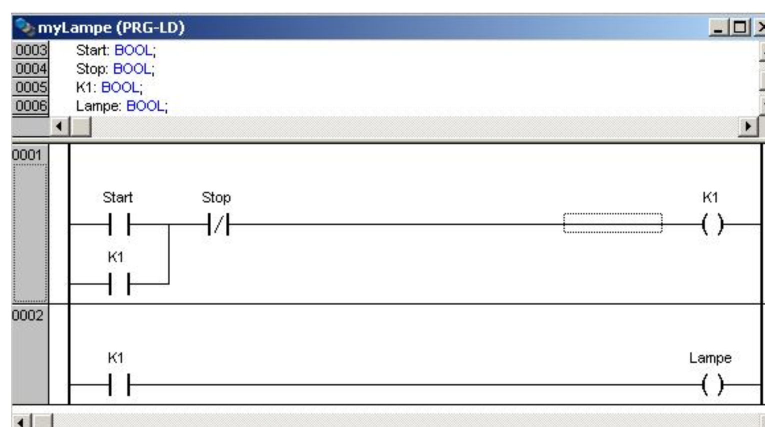

Fig. 2 Program in LD for the network in Fig. 1

The program written in Ladder is executed from top to bottom, network by network, except if this order is modified by jump instruction.
Variables in a network are evaluated **from left to right**. The left side of each network is composed of a series of contacts. It represents a logic function for which the value (TRUE/FALSE) is transferred to the right side, which is composed of coils or other output elements, such as Function Blocks.

Ladder Diagram is an ideal programming language for Boolean combinatorial logic equations, simple or more complex. It is also easy to integrate Function Blocks into the equation, for example: Timers and Counters.

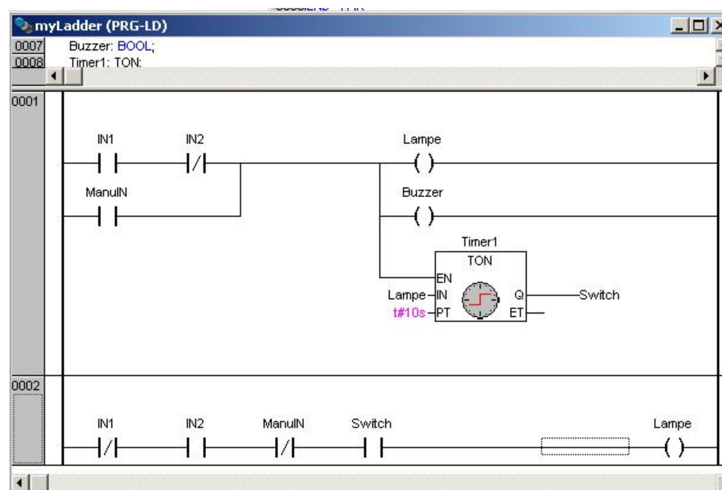

Fig. 3 A program in LD calling a Function Block

With Ladder, a simple representation of logical relationships between the Inputs, Outputs, and internal variables (in the form of circuits and contacts) is enough to start programming the application.

Thanks to the resemblance to electrical circuits, even a "non programmer" with basic knowledge in electrical engineering and logic can understand a program written in Ladder and, for example, identify and diagnose a problem or check a functionality of application.

Its basic concept and its programming style allows for almost all programmers, working in any country or industry, to read, understand and write the program using this language.

Therefore, Ladder Diagram is probably the most widely used programming language for logic control systems (PLCs).

However, similar to other PLC programming languages, LD programming requires a good preparation; this includes data organization, development of diagrams, and analyse of logical equations.

And if the complexity of the application increases, then it is often the challenge to keep the benefits and advantages of the Ladder programming: easy edition and coding, understanding and visualization.

Functions such as PID, data treatment, communication functions or even simple calculations, complicate programming in Ladder and are often difficult to implement.

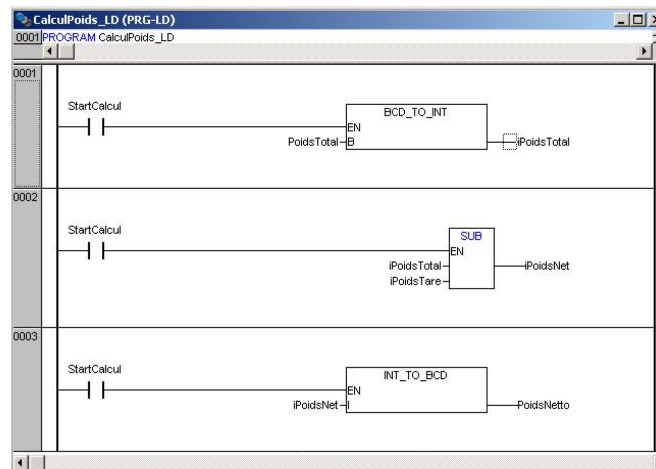

Fig.4 Example: Calculating weight in Ladder

Developing sequential programs in Ladder becomes another challenge. It is not a simple task to develop, review and diagnose a program in the form of a network of contacts. And analyse of the different stages of the process (states, steps) using input conditions and output activation, may quickly become very intimidating for an inexperienced programmer.

The program written in Ladder may become very difficult to read and interpret if it grows too large; even if the program is well written.

3. FBD: Function Block Diagram

The FBD language is a graphic language that is composed of foundational blocks that form a function between input variables (left) and output variables (right).

Input and output variables may be of a standard type (as is defined by the standard), or may also be of a type that is defined by the user (e.g. a table of variables, a structure, a pointer).

A block output may be connected to another block's input, on the condition that the "transferred" variables are of the same type.

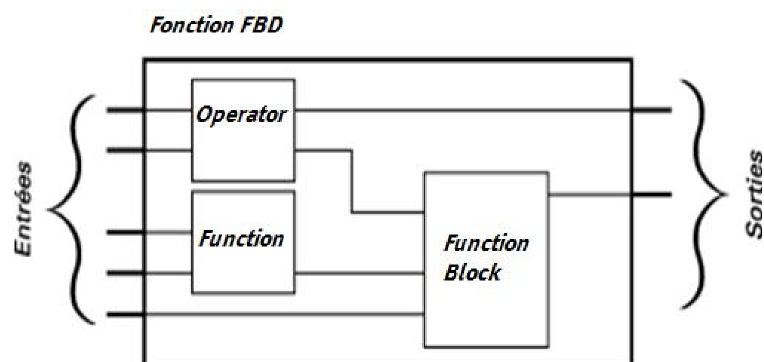

Fig. 5 The idea of FBD

The blocks composing a function in FBD can represent rather complex instructions:

- Standard operators, logic and arithmetic (AND, OR, ADD, SUB)
- Comparisons (LT, GT, NE, EQ, etc.)
- Mathematic functions (SIN, ABS, etc.)
- Function Blocks and Standard Functions defined by the standard
- User-created Function Blocks and Functions

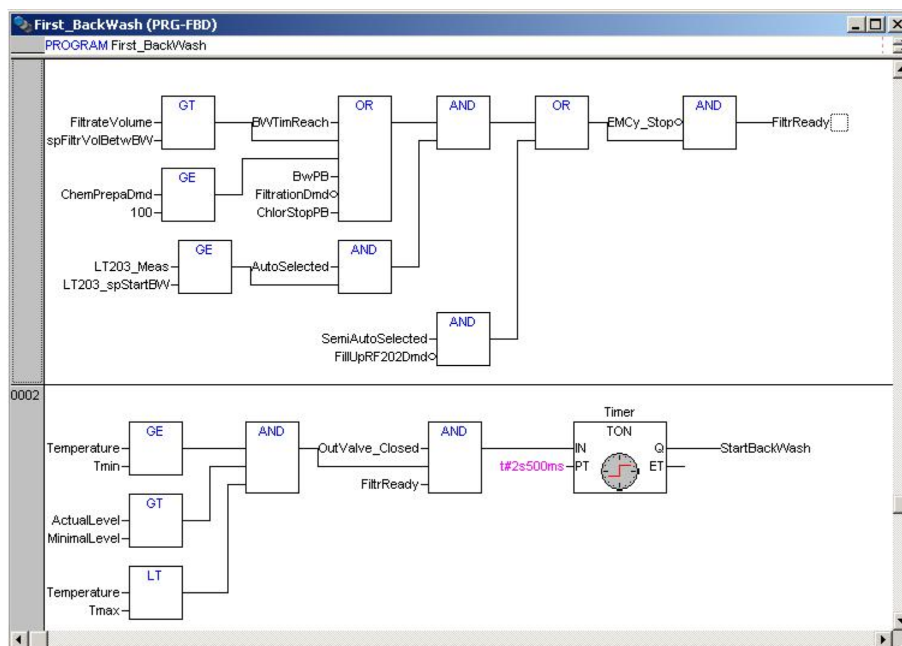

Fig. 6 Example of a program in FBD language

Compared to Ladder, an FBD function (network) may be visually easier to understand for users that are not really familiar with relay logic or electrical diagrams.

An FBD function forms a sequence that is easier to follow and diagnose than a network in LD. Just follow the evolutionary path of the variables!

However,

Similar to programs in LD, programs in FBD execute from top to bottom, function by function.

But, variables must be evaluated from the **output (right)** toward the linked block's **input (left)**; linked blocks being a Function or Function Block.

Thus, FBD functions are created and analysed **from right to left**, which is the opposite in Ladder.

The FBD language is rather well adapted for applications that treat several numeric variables using a simple numerical processing such as verification and scaling of analogue values, detecting when limits are exceeded, evaluating material balance, etc.

This explains why this type of programming is widely used in Europe, especially when programming applications for the field of chemistry (batch processes), or when basic calculations are required along with logic operations.

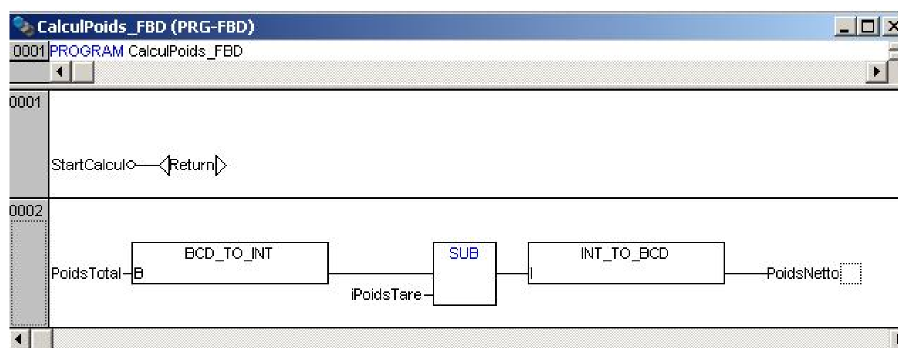

Fig.4 Example: Calculating weight in FBD

However, the FBD language is not recommended for programs that treat many variables or that call complex Function Blocks (several input and output variables).

If the size of a program written in FBD becomes important, if it contains too many variables and requires too much screen space for editing, then the program in FBD may quickly become too large to monitor and not easy to analyse.

Furthermore, developing a program in FBD requires more detailed preparation before writing the code:

- To understand the program and create the FBD diagram.
- To know how it will be executed (work)

Thus, considering the editing method for FBD functions, it may be difficult to subsequently make corrections.

Even though the use of FBD language has recently slowed, compared to other languages such as ST (Structured Text) and SFC (Grafcet), it is still probably the second most widely used language in the world of automatic control.

However, the application written in Ladder (LD) is always recommended for combinatorial logic functions.

4. IL : Instruction List

The IL language is a textual language similar to the assembly language. Its execution model is based on an "**Accumulator**" concept.

The Program in IL is made up of several lines of code (an instruction list) where each line (instruction) represents a single operation (command) to be carried out.

Each instruction contains an operator and one or several operands (separated by commas). The instruction may be followed by a label that identifies the line. A label may also be used as a jump address.

The IL language also supports comparative operators (EQ, GT, LT, GE, LE, NE) and jumps for programming loops or conditional program executions.

Jumps can be unconditional (JMP) or conditional (JMPC/JMPCN). Before executing conditional jumps, the accumulator value (TRUE or FALSE) is checked.

Fig. 8 Example of a code written in IL (instruction list)

Since IL language is a low level language, programs written in IL are compact and take up less controller memory space.

Also, programs in IL are executed more quickly than programs written in a graphic language, such as Ladder or FBD.

Contrary to graphic languages (LD, FBD), writing a program in IL doesn't requires a specific editor (mouse, clickable elements, etc.). Programs in IL can easily be edited with any text editor, even the most basic ones.

Furthermore, the IL file format has been standardised in XML by PLCopen. As a result, if the program uses only instructions defined by the IEC, then a program written in this language is transferrable (portable) and may easily be implemented by a different hardware platform.

Such strong points make this language very popular with European programmers, for whom transfer problems are often priority, unlike American programmers who prefer graphic languages that are easier to edit and diagnose.

It appears that technicians and service people prefer Ladder and FBD languages because they are more visual than the IL language and therefore easier to analyse and diagnose.

For them, speed and rapid execution are less relevant. Especially when taking into considerations a processing speed and the large amount of memory available on modern programmable controllers.

An inexperienced programmer may quickly be discouraged by the complexity of an application program written in IL language and which implements some complex functions like PID, communication routines, mathematic calculations, and complex logical equations (brackets!!!).

```
CalculPoids_IL (PRG-IL)
0001 PROGRAM CalculPoids_IL

0001    LDN     StartCalcul
0002    RETC                    (* retour, pas demande calcul *)
0003
0004    LD      PoidsTotal
0005    BCD_TO_INT
0006    SUB     iPoidsTare
0007    INT_TO_BCD
0008    ST      PoidsNetto      (* valeur poids nette *)
```

Fig.4 Example: Calculating weight in IL

Furthermore, the preliminary phase (analysis and writing the program in IL) is extremely important and must be properly carried out:

- Diagrams that illustrate how the program is to be executed
- Organization and definition of variables
- Comments!!!

The IL language supports jump instructions. Also it can be easily used for creating sequential applications (such as state machine, Grafcet).

But you must be vigilant to keep the program well-structured. Especially if you want to avoid the trap like "If GoTo" sequence. This could make the program difficult to analyze or debug.

Programming in IL demands a professional approach and is not recommended for beginner and/or inexperienced programmers.

So, for creating your IL program you must forget the Ladder/FBD programming style and the concept: *I program like I think*.

5. ST: Structured Text

Structured Text is a high level language. It involves many instructions, such as:

- Iteration loops (FOR; REPEAT-UNTIL; WHILE-DO)
- Conditions (IF-THEN-ELSE; CASE)
- Lines that end with semi-colons

Structured Text is quite similar to PC programming languages like PASCAL or C.

Structured Text is the IEC 61131-3 standard language which is better adapted for developing applications that require complex calculations, processing and analysis of mass quantities of data, communication functions and data exchanges.

```
TRANSMITTER_MEASURE (FB-ST)
0001 FUNCTION_BLOCK TRANSMITTER_MEASURE      (* without transmitter structure transfert *)
0002 VAR_INPUT

0001 (* ========================================================= *)
0002 (* =============== Faults/Alarms Timers ====================== *)
0003 (* ========================================================= *)
0004 (* Generalfa no inhibition on delay *)
0005     IF NOT BDI_GeneralfaInhib THEN
0006         BND_TON_GenfaTimer(IN:= BDD_StartTimerGenFa, PT:= BNI_spGeneralfaDelay, Q=> , ET=> );
0007     END_IF;
0008 (* HHfa *)
0009     IF NOT BDI_HHfaInhib THEN
0010         BND_TON_HHfaTimer(IN:= BDD_StartTimerHHFa, PT:= BNI_spHHfaDelay, Q=> , ET=> );
0011     END_IF;
0012 (* LLfa *)
0013     IF NOT BDI_LLfaInhib THEN
0014         BND_TON_LLfaTimer(IN:= BDD_StartTimerLLFa, PT:= BNI_spLLfaDelay, Q=> , ET=> );
0015     END_IF;
0016 (* Hal *)
0017     IF NOT BDI_HalInhib THEN
0018         BND_TON_HalTimer(IN:= BDD_StartTimerHal, PT:= BNI_spHalDelay, Q=> , ET=> );
0019     END_IF;
0020 (* Lal *)
0021     IF NOT BDI_LalInhib THEN
0022         BND_TON_LalTimer(IN:= BDD_StartTimerLal, PT:= BNI_spLalDelay, Q=> , ET=> );
0023     END_IF;
0024
0025 (* ====================== scaling of analog measure ============================ *)
0026 (* ScaledValue = [(spMaxScaleValue - spMinScaleValue)/(spMaxRawMeas - spMinRawMeas)]
0027                  * (RawMeasValue - spMinRawMeas)  + spMinScaleValue *)
0028 (* ========================================================================== *)
0029  IF NOT BDI_AnaChannelfa THEN
0030     BND_ScaledMeas := ((BNI_spfs - BNI_spls)/(BNI_spMaxGrossMeasure - BNI_spMinGrossMeasure));
0031     BND_ScaledMeas := BND_ScaledMeas * (BNI_GrossMeasure - BNI_spMinGrossMeasure);
0032     BNO_ScaledMeas := BND_ScaledMeas + BNI_spls;
0033     BND_LastScaledMeas := BNO_ScaledMeas;           (* memorisation od ScaledMeas *)
0034 END_IF;
```

Fig. 10 Example of code in ST (Structured Text)

Using iteration loops, conditions (decisions) and pointers (variables used for an indirect addressing) enables the development and implementation of really compacted programs. It is easier in ST to insert comments throughout the program and to use line withdraws or spacing when marking specific sections of code.

As such, it is easier to create a complex, well-structured program in ST than in other languages defined by the standard (Ladder, FBD or Instruction List).

This is all the more true especially since young engineers, recently graduated, have more experience with computer languages than with hard-wired logic basics. As a result, they quickly become more competent in Structured Text programming than graphic programming, which uses Ladder or FBD-type languages.

Moreover, the native machine code is denser in ST language, thanks to its non-graphic characteristics (like Instruction List). Thus, it executes more quickly than the same program written in Ladder or FBD.

Finally, programs written in Structured Text are easily transferred to other hardware platforms. Especially since "copying and pasting" text is really easy, and could be used to modify and/or adapt to a new platform.

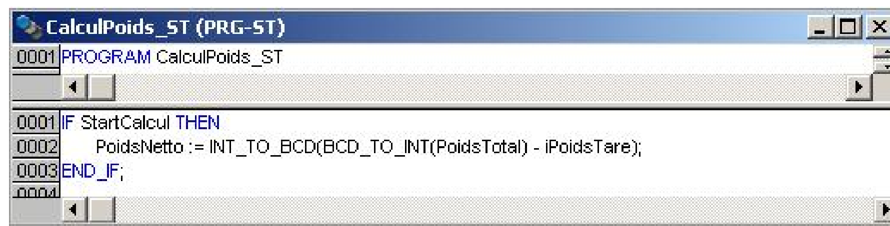

Fig. 11 Example of calculating weight in ST

However, do not forget that **Structured Text language does not support jump instructions!!**

This is a trap for beginners or, for example, for those who try to shift directly from graphical programming into Structured Text programming.

Another disadvantage of ST is the ease with which the user can develop complex programs. For a large number of programmers (even experienced) and for maintenance and service personnel, the Structured Text environment is not well-adapted for diagnosis or repair. For example, it is difficult to follow how sequential programs are executed: evolution of the steps, associated actions, transitions, input/output conditions, etc.

The same applies to complex logic equations (brackets).

Thus, a good idea could be to develop applications that benefit from structure and code density of ST program, and the ease of interpretation provided by graphic languages (LD, FBD, SFC).

Consequently, it is advised to use Structured Text as a base for developing Program Organisation Units (POUs: Functions, Function Blocks, Programs). Then, to implement these POU's within the program written in a graphic language (LD, FBD, SFC).

But, this requires a good programming discipline and well completed preliminary phase. The developer-programmer must test and debug his POU code well, because the programmer-user of the application will probably not have access to the code. Or, more simply, he won't be sure enough of his skills to attempt any modifications...

Furthermore, the preliminary phase (analysis and writing the program in ST) is still nonetheless extremely important and must be properly carried out:

- Diagrams that illustrate how the program is structured and will be executed
- Organization (data type) and definition of variables
- Comments!!!

6. SFC: Sequential Function Chart

The Sequential Function Chart (SFC) is a graphic programming language for Industrial PLC's and is defined by the IEC 61131-3 standard. It has been defined by PLC Open after "French lobby" intervention for integrating the French GRAFCET specification into the 61131-3 standard.

Remember:

- The GRAFCET is a specification (NF C 03-190, EN 60848); a way to represent the behaviour and analysis of an automated system, specifically well-adapted to sequential evolution systems; which means easily broken down into steps.
- **SFC is a programming language** (graphic) **for coding an automatic control application's GRAFCET** and that will be **executed by a controller**.

As a result, the SFC language is a fairly loose interpretation (more permissive) of the GRAFCET, which allows the user to quickly develop and program operation sequences and easily validate the program in its entirety (before implementation).

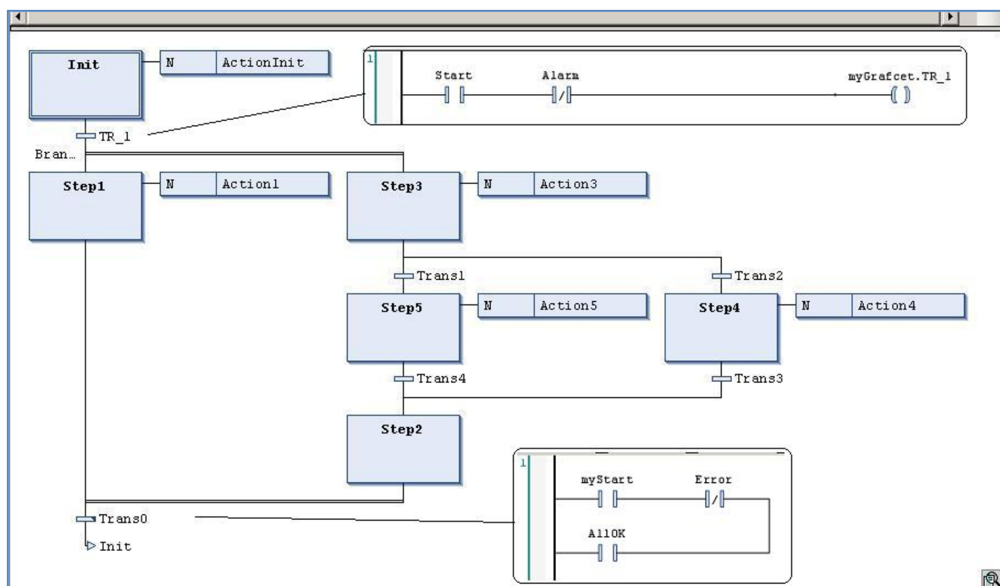

Fig. 12 Example of a program in SFC language

The SFC concept remains basic, similar to GRAFCET and its set of evolution rules. A step in a series of sequences is linked to the actions and transitions. The actions are carried out when the step is activated.

If the step is activated and the transition that follows is validated (true condition) then the step is immediately deactivated, the transition "passed" and the following step is activated.

If the step is no longer activated, then the actions linked to that step won't be carried out (unless... see details on GRAFCET rules).

SFC actions are programmed in any standard language, including SFC.

Transitions may be programmed in LD/ST/FBD/IL in the form of a Boolean function.

The SFC language is particularly well adapted and easy to implement for sequential applications that may be broken down into steps or a series of repeatable processes.

So, a program written in SFC is made up of POU-SFC (GRAFCET) and of several simple routines (codes) for actions and transitions linked to the step.

However, following GRAFCET evolution rules (although there are exceptions):

- There is a generally only a part of the code that is active for the actions of the activated step.
- And the control of the program flow could be made by a simple verification of the transitions linked to the activated step in the program sequences.

Embedding and coding complex sequences is easy in SFC. For example, an application's stop and start modes (e.g. French “GEMMA” concept).

This is why maintenance personnel and beginner programmers like using SFC; because its visual characteristics and code segmenting allow them to precisely monitor all of the states, find "blocking points" and easy repair the application.

It should also be noted that the ISO 1219-2 standard recommends that SFC should be used for “programmable” automation project documentation.

This is because SFC can complete the project documentation, helping and assisting the programming engineers, client-users and maintenance personnel to understand how the equipment operates and how the application functions.

The downside:

The programming style in SFC is not always suitable for application which require complex numeric processing such as communication tasks or dialog with MMIs. Programming with SFC could add unnecessary complexity that is difficult to encode.

It should also be remembered that SFC is a programming language that does not fully respect GRAFCET specifications. This is seen as inadmissible by French purist GRAFCET proponents.

A particular example is the forcing of a step or the processing of transitions in the case of divergences in OR (in SFC transitions are sequenced and processed with a priority, as exclusives – in contrast to GRAFCET).

Furthermore, GRAFCET’s different modes of visual representation (and its coding) do not make programming easy. For example, in SFC, step related actions can be initiated in three modes:

- On activation of the step (input actions)
- “Conventional” actions associated with the active step
- On deactivation of the step (output actions)

On the other hand, it should not be forgotten that SFC is a graphics language used to “encode” GRAFCET. As a result, SFC programming requires thorough preparation and planning of work. For example, while encoding (drawing the GRAFCET, programming of actions and transitions) can be performed without any major difficulties, the modifications and corrections can be difficult to handle and monitor.

Furthermore, greater resources (in memory and speed of execution) are required for storing and running the SFC program. Programs written with SFC run with pre-processing (transitions) and post-processing (steps and actions). Also their execution could be slower than programs written in other languages.

Additionally, there is one last, but significant, shortcoming of the SFC language.

SFC is the only language in the standard which cannot be converted into other languages proposed by the standard IEC 61131-3.

Programs and function blocks written with IL, LD and FBD can easily be converted into another language, allowing coding and display of the application in the most comfortable manner for the user.

The same is true of POU's written in ST (Structured Text) which can be transformed into any of these three programming languages (IL, LD, FBD).

As a result, it is recommended to implement this language for users who know GRAFCET and where the encoding/display format of the application is unlikely to change.

Conclusions: choose a language suited to your project

With the various programming languages proposed by the standard IEC 61131, it is difficult to provide you with precise advice. However, it is important to take into account several factors before deciding which one you wish to use for your application:

- The kind and complexity of the application.
- The requirements and practices of the end customer and its maintenance dept.
- The time remaining for implementation.
- Follow-up of the project.
- Etc.

Of course, if you are already familiar with any language, if you already have tried and tested programming and implementation practices, it would be best to recommend that you stick with what you know and are comfortable with.

Lastly, the PLC or the programming platform can also influence your choice of programming languages.

However, before you start, take a look at the table below…

Don't hesitate to make any comments!

Language	Strong points	Weak points
LD	- Recognised in the automatic control world - Boolean and binary switching functions - Quick processing (optimised cycle time) - Easy integration of function blocks (standards, user) - Program visualization, interpretation and trouble-shooting - Easy to modify	- Mathematical calculations (even simple) - Data processing (strings of characters, analog I/O, communication routines) - Repetitive instructions, loops - Sequential logic with a significant number of sequences - Creation of user function blocks with a wide range of variables
FBD	- Boolean switching functions - Simple mathematical calculations - Analogical data processing - Easy integration of function blocks (standards, user) - Program visualization, interpretation and trouble-shooting	- Sequential logic with a significant number of sequences - Repetitive instructions, loops - Creation of user function blocks with a wide range of variables - Programme modification - Viewing and interpretation in the case of complex programs (loops, jumps)
IL	- Quick coding and data entry (simple text) - Compact code - Fast processing and execution speed (optimised time cycle) - Repetitive instructions, loops, jumps - Simple sequential logic - Slim viewing and display - Portability, transferable to other platforms	- Complex Combinatorial logic - Mathematical calculations and data processing - Difficult to follow after use by someone else (interpretation, comprehension, modification) - Interpretation and diagnosis by inexperienced users
ST	- Quick coding and data entry (simple text) - Programming structured like a high level PC language - Compact code and good execution speed - Mathematical calculations and data processing - Repetitive instructions, loops - Ease of use and take-up by newcomers - Creation of user function blocks with a wide range of variables - Portability, transferable to other platforms	- Combinatorial logic with a significant amount of variables - Heightened programming discipline - Lack of jump instructions - Danger of large or endless loops (watchdog error) - Visualisation and interpretation in the case of complex programs with a wide range of variables - Modification of this type of program
SFC	- Easy realisation of sequential applications and repetitive tasks - Clarity and accurate presentation of sequences - Embedding of complex sequences (e.g. GEMMA) - Easy to maintain by the end user - Project documentation help	- Complex data entry and programming (graphics, actions, transitions) - Cannot be converted to other languages in the standard - Not fully compliant with GRAFCET (forcing of steps, different action modes, OR divergence processing, etc.), inadmissible for GRAFCET purists - Requires greater level of resources

Tab. 1 My subjective point of view on the languages in the IEC 61131-3 standard

Appendix 1: Summary of advices found on the web:

- Easy to maintain by the end user: SFC
- Widely accepted in the automatic control world: LD
- Accepted in Europe: FBD, LD, IL or ST
- Execution speed on API: IL or ST
- Applications with numeric I/O and basic processing: LD or FBD
- Easy to monitor and change code: LD
- Easy to use by new users: ST
- Implementation possible of complex mathematical operations: ST
- Applications with sequential processes: SFC

CoDeSys : Five languages of IEC 61131-3 in the Automation Project

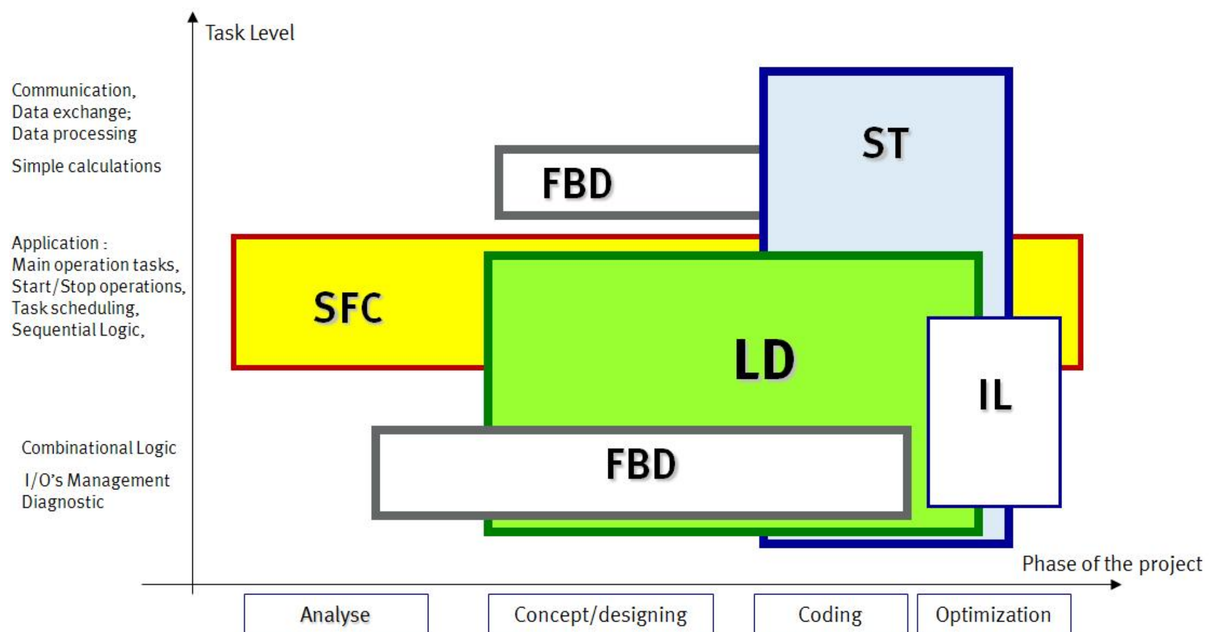

Fig. 13 Suggested use of the languages in standard IEC61131-3

Appendix 2: CFC: Continuous Function Chart

The CFC language is not officially standardised by the IEC 61131-3 standard. However, it is often proposed (e.g. by Siemens, B & R, CoDeSys, KW-Multiprog) as an extension of the standardised language FBD.

Like FBD, CFC is a graphics language which resembles electronic circuit diagrams. The function to be performed is presented in the form of interconnected graphic elements. CFC graphic elements include blocks, inputs, outputs, jumps, returns, tags and comment boxes.

However, the main difference between these two languages is the editing of a POU (program, function block).

FBD is network oriented (like LD) where each element (block or link) of a function is placed and connected automatically by the language editor. As a result, it is impossible to create (explicit) return loops. Instead, an intermediate variable should always be used.

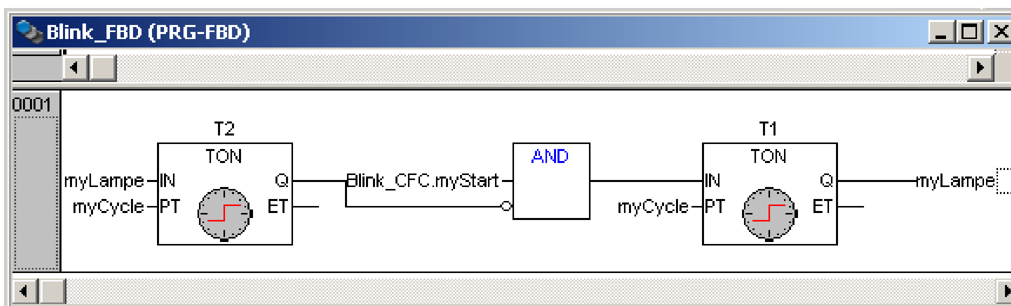

Fig.14a Example of a “Blinker” programme in FBD

With the CFC editor, all the graphic elements can be freely placed on the editing sheet. The inputs and outputs of these elements can then be interconnected by the user (simply by dragging and dropping with the mouse). The CFC editor automatically determines and builds the paths to be followed, even across the limits of the editing page. Furthermore, if the graphic elements are moved by the user, the connection lines are automatically adjusted.

Of course, the loops can be created directly, without intermediate variables.

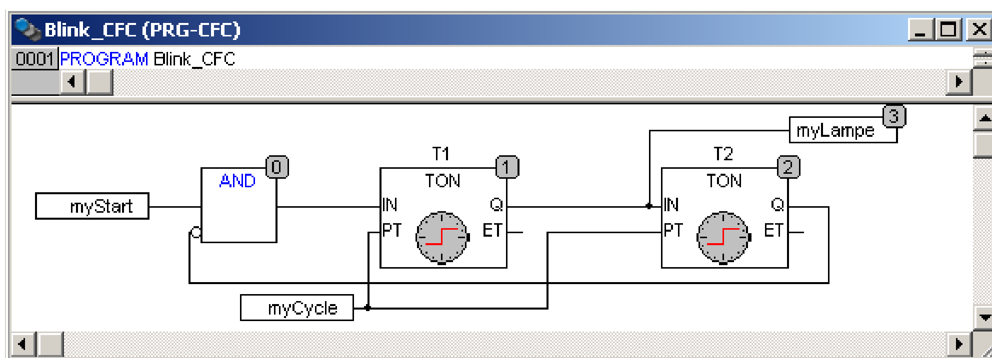

Fig.14b The same “Blinker” program in CFC

More, in CFC, only the “necessary” connections must be set up between elements. This simplifies readability of the CFC program, **which is especially well suited for providing a graphic preview of an application** (maintenance!).

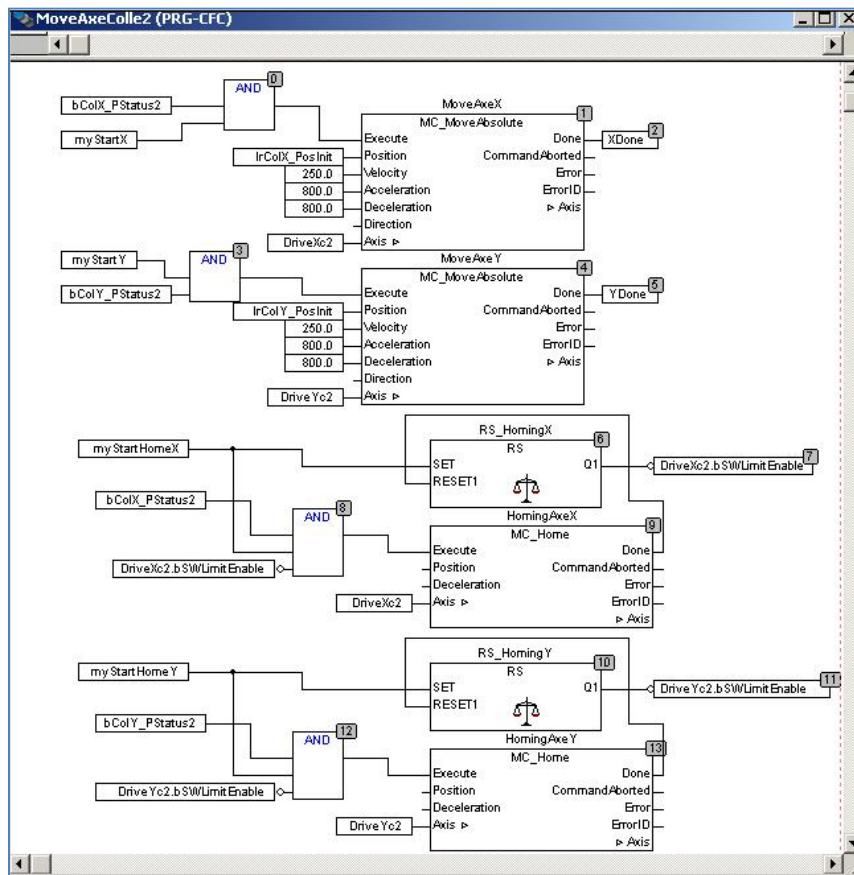

Fig. 15 Example of an application in CFC

The sequence in which the elements of a CFC POU are executed is indicated by the numbers in the upper right-hand corner of the element. Processing starts with the element that has the lowest number, i.e. 0.

And if an element is added, its number will automatically be adjusted according to the topological order of the editing sheet: from left to right and top to bottom.

The new element will be assigned the number of its predecessor “plus 1” and all the higher numbers will be increased by ONE.

The execution order can be modified by the programmer and it is important to note that the execution sequence can influence the computed result. This is particularly the case for complex structures with loops.

It should also be pointed out that if the element is moved, without alteration of connections, the element number (execution order) does not change.

The main advantage of CFC is its **easy programming**. Pre-defined blocks only need to be connected to one another and then configured (if necessary).

As a result, creation of programs simply by interconnecting standard elements is quicker and less “sensitive” to errors than conventional programming. Therefore, in theory, extensive experience in programming is not necessary.

However, there are also disadvantages.

Firstly, CFC is not standardised by the IEC 61131-3 standard, so it is likely that problems will be encountered when transferring to other platforms. Nevertheless, it is often possible (e.g. CoDeSys) to convert a CFC POU into another language of the standard.

In the case of complex programs or blocks with a wide range of input/output variables and loops, the readability of CFC programs is open to debate. This is especially true for programmers who intervene within the application to modify it or for maintenance personnel (display in on-line mode).

Printing on paper can also be problematic (e.g. documentation for users).

Finally, if CFC is very suitable for applications with arithmetic functions, combinatorial logic and simple calculations, it is not very suitable for sequential logic.

Wojciech Gomolka
FESTO France
Bry sur Marne
July 2011