PERGAMON

Control Engineering Practice 6 (1998) 547–555

Implementation aspects of the PLC standard IEC 1131-3

Martin Öhman*, Stefan Johansson, Karl-Erik Årzén

Department of Automatic Control, Lund Institute of Technology Box 118, S-221 00 Lund, Sweden

Received 8 October 1997; accepted 9 February 1998

Abstract

IEC 1131-3 is a standard for PLCs, defining four programming languages and a type of Grafcet, Sequential Function Charts (SFC). An object-oriented prototype of SFC and the language Function Block Diagram have been implemented. Various execution methods are discussed. Algorithms for local and global sorting are implemented and evaluated. The standard is found to be unclear in some parts.

Keywords: Grafcet; IEC 1131; programmable logic controllers; standards

1. Introduction

Programmable logic controllers (PLCs) are specialized computers, widely used in industrial automation. IEC 1131-3 is a standard that defines four programming languages used in PLCs. This paper is concerned with the function block diagram (FBD) and sequential function charts (SFC) parts of the standard. It is based on a project (Johansson and Öhman, 1995) where the purpose was to make a prototype implementation of the standard and evaluate different implementation aspects. The standard is described in Section 2. The order of execution between function blocks and SFC is discussed in Section 3. In Section 4 the prototype implementation is described. Different implementation aspects and alternatives are discussed. An example is shown in Section 5.

2. IEC 1131-3

This section describes IEC 1131 (IEC. 1995b), a PLC standard defined by the International Electrotechnical Commission (IEC). The standard contains five parts. This paper is concerned with part three, describing the programming languages.

IEC 1131-3 defines four different language paradigms, SFC, and program organization units. The standard also specifies their representation, and rules of evaluation.

*Corresponding author. Tel. +46 46 222 03 62; e-mail: martin@control.lth.se

2.1. Programming languages

The four defined language paradigms are described in this section. They are illustrated with an example, written in all four languages.

Instruction list. The instruction list language is similar to assembly code, with commands like LOAD and STORE. An accumulator is used to store results. This language corresponds to the programming technique that has traditionally been used in PLCs. The instruction list language is shown in Fig. 1.

Structured text. The structured text language is similar to Pascal. It has sequential statements, conditional statements like IF and CASE, and repetitive statements like FOR ... DO ... END_FOR and REPEAT ... UNTIL. The structured text language is shown in Fig. 2.

Ladder diagrams. The ladder diagram language specifies how to use relay ladder logic diagrams to implement boolean functions. This is a common language in modern PLCs. The ladder diagram language is shown in Fig. 3.

Function block diagram. In the function block diagram language, all functions, inputs and outputs are represented as graphical blocks. They are connected by lines representing the data flow. The direction is always from left to right, except in feedback paths. The function block diagram language is shown in Fig. 4.

0967-0661/98/$ – See front matter
PII S0967-0661(98)00054-9

```
FUNCTION D : BOOLEAN

  VAR_INPUT
    A, B, C : BOOLEAN;
  END_VAR

  LD A
  OR B
  AND C
  ST D

END_FUNCTION
```

Fig. 1. Instruction list.

```
FUNCTION D : BOOLEAN

  VAR_INPUT
    A, B, C : BOOLEAN;
  END_VAR

  D := A OR B AND C;

END_FUNCTION
```

Fig. 2. Structured text.

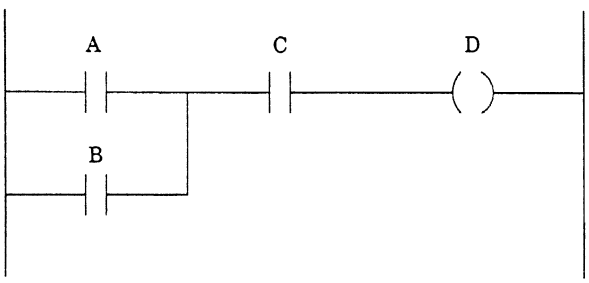

Fig. 3. A ladder diagram.

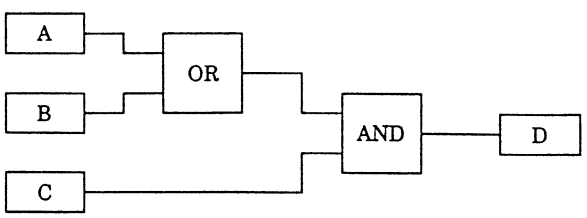

Fig. 4. A function block diagram.

According to the standard, it should be possible to do jumps in all the languages. It is probably not so good to do jumps in the FBD language. The guidelines (IEC, 1995a) also advises against that.

2.2. Program organization units

Independent of the choice of language, PLC programming according to IEC 1131-3 uses three program organization units: functions, function blocks and programs. They are supplied by the manufacturer, or defined by the user. A unit cannot be recursive, i.e., it cannot call itself. Fig. 5 shows how the units function together.

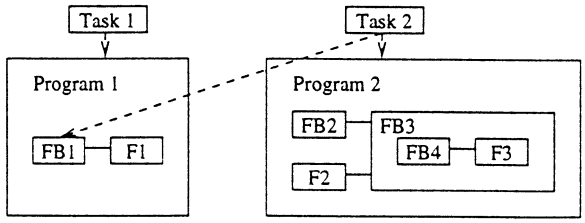

Fig. 5. Program organization units.

Functions. Functions have one or more inputs but only one output. A function cannot store any state information, i.e., executing a function with certain values on the inputs always gives the same output value. Functions can be hierarchically defined by using already defined functions. A function can be *extensible*, meaning that the user can decide the number of inputs. The standard does not state if the user has to specify the number of inputs when the function is created, or if the number can be changed during the lifetime of the function. Functions are either *typed* (operate on a specified data type) or *overloaded* (operate an different types).

Function blocks. Function blocks can have several inputs and outputs. They can store state information, so the values of the outputs depend on previous values.

The user must give each instance of a function block a unique name. Function blocks can be hierarchically defined by using already defined functions and function blocks.

The standard specifies that there can be multiple instances of function blocks. Functions can probably also have multiple instances.

Programs. Programs are built up of functions and function blocks. Programs do not have inputs or outputs.

2.3. Tasks

The execution of programs and function blocks is controlled by tasks. The standard does not specify whether the scheduling of tasks should be preemptive or non-preemptive. A task can control more than one program organization unit, and a program organization unit can indirectly be controlled by more than one task, as FB1 is in Fig. 5. If they try to execute at the same time and if preemptive scheduling is used, the implementation must ensure that manual exclusion is obtained.

A task has three inputs: single, interval and priority. It can execute once on the rising edge of *single*, or

periodically with the period time of *interval*. The priority of the task is set by its *priority*.

2.4. Sequential function charts

A sequential function chart (SFC) is an extended state machine, also known as a Grafcet (David and Alla, 1992). In the standard, SFC is a way of structuring programs and function blocks. A unit that is not structured is said to be a single action, executed continuously.

An SFC consists of two main elements, *steps* and *transitions*, shown in Fig. 6. Steps can be active or inactive. The state of the SFC is determined by which steps are active. A transition has a Boolean input, the *transition condition*, that can be described by any of the four languages or by a Boolean variable. A transition will fire when the step above it is active and the transition condition is true. SFC in IEC 1131-3 does not support macro steps, as Grafcet normally does. An SFC can contain parallel and alternative paths, shown in Fig. 6.

Actions. It is possible to associate an *action* with a step. An action can be a single Boolean variable, can be described using one of the four programming languages or can be described by an SFC. All actions must have unique names.

Action blocks. Action blocks are used to associate actions with steps. An action block has a Boolean input that indicates when the associated step is active. Each action block is associated with an action. Steps, actions and action blocks relate in the following way:

- A step can be connected to one, zero or more action blocks.
- Each action block is connected to one step.
- Each action block is associated with one action.
- Each action is associated with one or more action blocks.

Fig. 7 shows that the Boolean action A2 is associated with the action blocks AB1 and AB3. The non-Boolean action A1 is constructed directly in the action block AB2, and associated only with that action block.

Action qualifier. An action block uses an *action qualifier* to control the action. The action will execute, depending on the action qualifiers on the associated action blocks and the status of the associated steps. The different action qualifiers are shown in Table 1.

Fig. 7 shows action blocks, action qualifiers and actions.

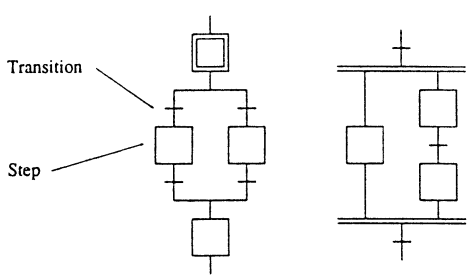

Fig. 6. SFCs with alternative (left) and parallel (right) paths.

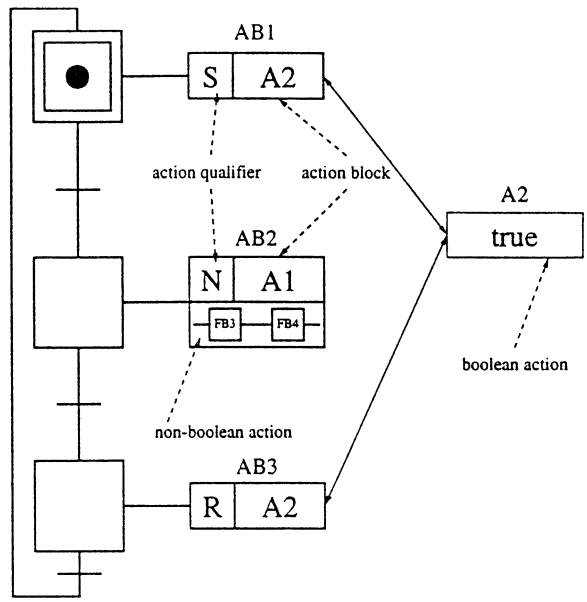

Fig. 7. An SFC with action blocks, action qualifiers and actions.

Table 1
Action qualifiers

N	**Non-stored** The action is active while the action block is active.
S	**Stored** The action is activated when the action block becomes active.
R	**Reset** The action is deactivated when the action block becomes active.
L	**Limited** The action is activated when the action block becomes active, and is deactivated after a certain time or when the action block becomes inactive.
D	**Delayed** The action is activated a certain time after the action block becomes active, and is active as long as the action block is active.
P	**Pulse** The action is active for one clock cycle when the action block is activated.
SD	**Stored and delayed** The action is activated a certain time after the action block becomes active.
DS	**Delayed and stored** The action is activated a certain time after the action block becomes active, if the action block is still active.
SL	**Stored and limited** The action is activated when the action block becomes active, and is deactivated after a certain time.

3. Execution order

This section deals with the execution order between functions and function blocks, SFC-elements, action

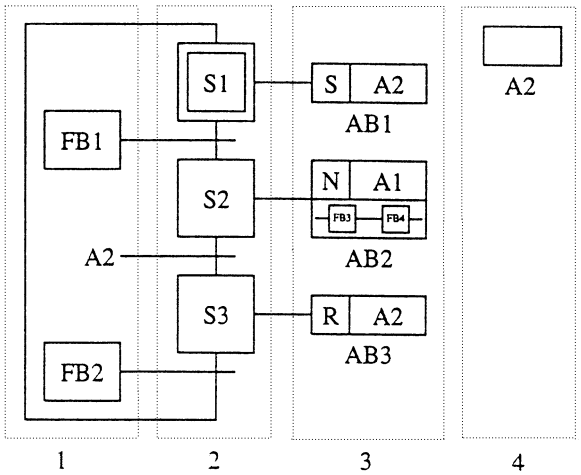

Fig. 8. The execution order of functions, function blocks, steps, transition, action blocks, and actions.

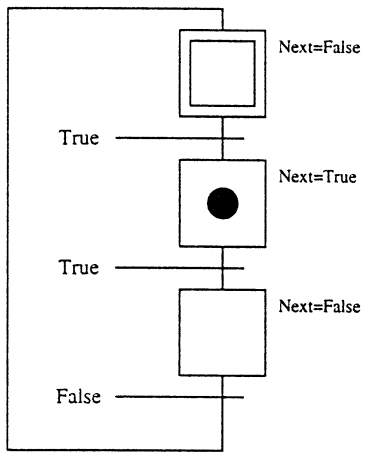

Fig. 9. Execution of an SFC.

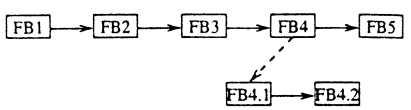

Fig. 10. The function block list of the function block diagram in Fig. 14 and the local list of FB4.

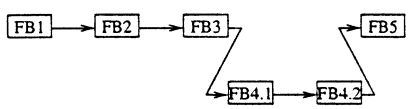

Fig. 11. The global function block list of the function block diagram in Fig. 14.

blocks, and actions. This is not well specified in the standard. It is stated that no element shall be evaluated until the states of all its inputs have been evaluated. The authors believe that this leads to the following execution order, shown in Fig. 8:

1. Function blocks connected to a transition.
2. Steps and transitions.
3. Action blocks and non-Boolean actions.
4. Boolean actions.

Another idea would be to execute steps and action blocks in the same phase, executing only action blocks connected to active steps. This method would, however, cause problems with, or example, the action qualifier "stored". Then the action block will also be executed after the step has been deactivated.

3.1. Sequential function charts

The standard specifies that all steps should be updated synchronously. This could be achieved by going through all the steps twice. First, it is checked which steps can be activated, and they are marked with a flag "next", see Fig. 9. Then all marked steps are activated. The order in which the steps are treated is not important. This method makes sure that each token can pass only one transition at a time.

3.2. Function blocks

Functions and function blocks are executed so that the dataflow goes from left to right. One way of storing the determined execution order is to insert the functions and function blocks in a dynamic list. The standard states that functions and function blocks can be hierarchically defined. It is not specified in the standard if such a function or function block should be considered as a fixed unit, or if its internal structure can be changed. If a unit has a hierarchical structure, the execution order can be dealt with in two ways: local and global sorting. It is not specified which one should be used.

Local sorting. In local sorting, each hierarchical function block have its own list, containing the functions and function blocks in its internal structure. Fig. 10 shows the main function block list of the FBD in Fig. 14, and the local list of FB4.

The main advantage with local lists is that if changes are made in a hierarchical function block, only a new local list for that function block has to be built. This corresponds to the separate compilation of program units.

Global sorting. In global sorting one has a global function block list for each top-level function block containing function blocks at all levels. No hierarchical function blocks are inserted in the lists; only the function blocks on their subworkspaces. Fig. 11 shows the global list of the FBD in Fig. 14. Note that FB4 is not in the list.

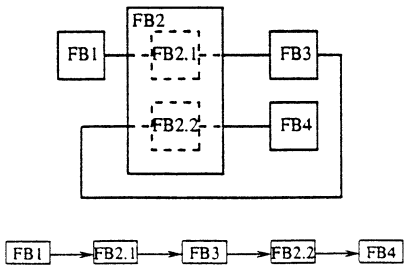

Fig. 12. A loop is removed by using global lists.

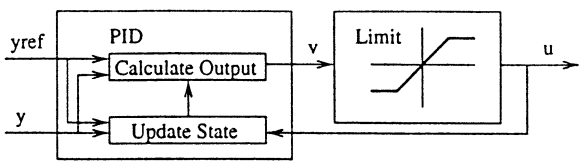

Fig. 13. A PID controller.

The main advantage with global lists is that imaginary loops that seem to exist from a local point of view are removed. If the function block diagram in Fig. 12 were to be sorted locally, the user would have to insert a loop element (described in Section 4.3). The global list shows that this is not needed with global sorting. Removing loops gives you a faster implementation, by saving clock cycles.

A disadvantage with global lists is that the implementation is more complex. When a hierarchical function block is disabled, all the function blocks in its internal structure should be disabled. If they are spread out in a global list, this will be difficult.

An example. PID controllers are commonly used in industrial control systems. In IEC 1131-3 a PID controller is naturally implemented as a function block that has reference signal y_{ref} and measured variable y as inputs, and the control signal v as output. However, if actuator saturations are not taken care of in the right way, the result is integrator (reset) windup, which negatively influences the control performance. A common solution to integrator windup is known as "tracking". The approach is based on feeding the saturated control signal back to the PID controller and to changing the integral term in the controller in such a way that the control signal generated from the controller will equal the output saturation.

Hence, tracking requires an additional input to the PID controller: the saturated output, often denoted u. Tracking also means that it is not possible to update the controller state until the saturated output is known, e.g. has been calculated. Sometimes the saturation limits are known in the PID controller, but not always. For example, the PID controller may be a part of a cascade construct, or the actuator saturation is calculated in a special limit block, as shown in Fig. 13.

The PID function block is internally decomposed into two blocks: CalculateOutput and UpdateState. The only sorting strategy that will give the correct behavior from a control point of view is global sorting. Using that strategy the execution order will be PID: CalculateOutput, Limit, PID: UpdateState. Using local sorting, the old value of u would be used in the calculation of UpdateState, i.e. an unnecessary delay of one sampling interval would be introduced.

3.3. General execution algorithm

Combining the discussions above gives the following general execution algorithm:

- For each function or function block (sorted locally or globally):
  - Read inputs.
  - Execute function block.
  - Write outputs.
  - Propagate output to connected function blocks.
- For each step:
  - Check if the step can be updated.
- For each step:
  - If the step can be updated,
    *Update the step.
  - Write to action block.
- For each action block:
  - Determine if the associated actions should be active.
  - Possibly execute associated non-Boolean actions.
  - Write to Boolean actions.
- For each Boolean action:
  - Update the action.

4. Implementation

In this section the prototype implementation is described. The implementation is done in G2, a graphical and object-oriented programming environment from Gensym Corporation. G2 was chosen because it is good as a rapid prototyping tool. Different implementation methods are discussed. Most of the ideas can be used generally, when working in an object-oriented environment. Some solutions, however, are G2-specific.

A hierarchy of classes containing tasks, function blocks, connections, steps, transitions, action blocks, actions, and global variables has been developed. The user constructs the programs on a workspace with graphical objects that are instances of these classes. Class definitions and methods are hidden from the user.

4.1. Function blocks

Since programs and functions can be seen as special cases of function blocks, only *function blocks* have been

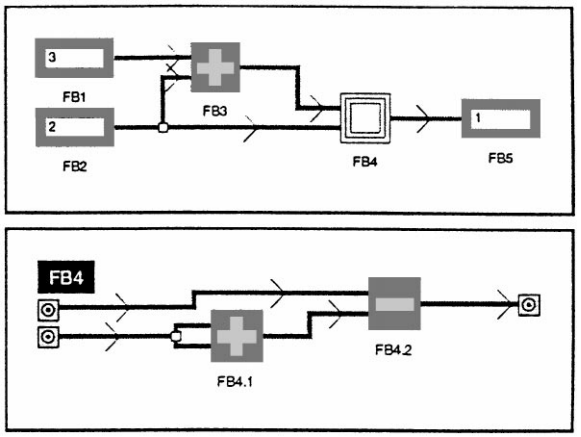

Fig. 14. A hierarchical function block diagram.

implemented. Programs are implemented as function blocks with no inputs or outputs, and function as single-output function blocks without internal state information. Fig. 14 shows a function block diagram.

To determine the order of execution between function blocks, a list is built when a task is initialized. The function blocks are inserted in the list in the correct execution order. Each function block is placed in the list after the function blocks connected to its input.

4.2. Hierarchical function blocks

Hierarchical function blocks have a subworkspace where their internal structure is defined; see FB4 in Fig. 14. G2 automatically makes a link from the function block to its subworkspace. It would be natural to include the internal structure in the class definition, but this is not possible in G2. Hierarchical function blocks are inserted in execution lists using either local or global sorting.

4.3. Algebraic loops

The order in which the function blocks are executed must follow specific rules. A function block cannot be executed until the function blocks connected to all its inputs have been executed. This rule causes problems when building a loop. In Fig. 15, FB1 must be executed before FB2 to give the input of FB2 a value. For the same reason, FB2 must be executed before FB1. This problem is called an *algebraic loop*.

The problem is solved by introducing a so-called *loop element*. The user decides where to break the loop and insert a loop element. A loop element has the special property that it can be executed before function blocks connected to its input. It reads the old value from the input and writes it to the output. Therefore, the user can use them, instead of variables, as a memory. In Fig. 16 the algebraic loop problem is solved by inserting a loop element, and FB1 can now be executed before FB2.

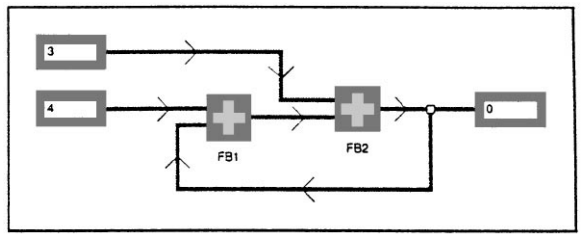

Fig. 15. Algebraic loop.

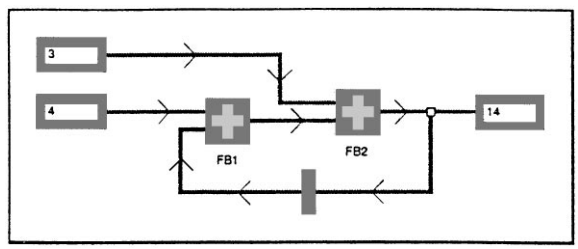

Fig. 16. Loop elements.

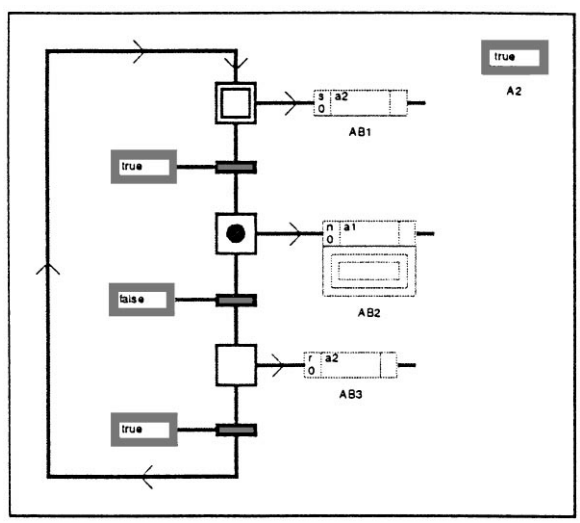

Fig. 17. An SFC with steps, transitions, action blocks and actions.

4.4. Sequential function charts

SFC elements include steps, transitions and branches. Branches are used to build alternative and parallel paths. All the elements have methods, used to determine whether a step should be activated or deactivated.

All steps and transitions have an input and an output, used to connect them with each other. Steps are represented graphically as rectangles. A filled red circle indicates when a step is active. The state of the step is written to a Boolean output that can be connected to an action block. Transitions are represented graphically as a horizontal bar. They have a Boolean input that acts as a transition condition. Fig. 17 shows an SFC.

4.5. Global variables and actions

Global variables of different data types are implemented. Actions are a subclass of Boolean variables. To get the value of a variable, special *get-var* functions are used. If the action only functions as a boolean variable it has a graphical representation and is placed on an arbitrary workspace by the user. If the action is defined with function blocks or SFC, however, it is constructed on a subworkspace of that action block, and can then be associated only with that action block. The standard defines when an action should be active, depending on the action qualifiers of the associated active action blocks. This functionality is implemented in a method of the class action.

When an action is defined on the subworkspace of an action block, the action block functions like a hierarchical function block with an enable signal associated with the action. The function blocks on the subworkspace should only be executed when the action block is enabled. This is solved by placing those function blocks in local lists, even when the rest of the resource is sorted globally, i.e., total global sorting is not implemented.

4.6. Tasks

To control the tasks, a *task handler* is defined. At each time unit the task handler decreases a counter in each task. When the counter reaches zero, the task will execute and reset the counter to the execution interval. When a task executes, it runs all its associated function blocks. Hence, a link from the task to the associated function blocks is needed. A natural solution would be to let the task have a static list of pointers to all its function blocks, but G2 does not support pointers. Instead, each function block stores the name of its task, and when a task is activated G2 effectively inserts all the associated function blocks in a list.

4.7. Execution procedure

Execution of elements could also be handled by replacing the lists with a procedure that would be generated automatically when the task is activated. The list would contain the code from all the function blocks, inserted in the correct order, using either local or global sorting. Each time a task is executed it would only execute the procedure. A small example was made to evaluate the time difference, see Fig. 18.

The execution time between a function block list, (Fig. 19) and a procedure, (Fig. 20) when executed 100 000 times, were compared. Execution with the list took 112 s, and with the procedure 7 s. This shows that there is a lot to be gained by using procedures.

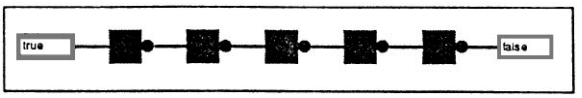

Fig. 18. A test example.

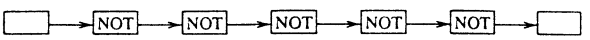

Fig. 19. The function block list of the test example in Fig. 18.

```
begin
   x = true;
   x = not (x);
   x = not (x);
   x = not (x);
   x = not (x);
   x = not (x);
end;
```

Fig. 20. The procedure of the test example in Fig. 18.

4.8. Asynchronous dataflow

Another solution would be to let the execution by asynchronously controlled by the data flow. Each function block would then be executed as soon as the value of any of its inputs was changed. This does not correspond well to the standard, since it is stated that each function block should be executed in every task cycle, independently of whether its input values have changed or not.

5. An example

This section describes how FBD and SFC could be combined in a larger and slightly more realistic example. The process is simulated in G2. Fig. 21 shows the process, which contains a tank, a pump, a valve and a heater.

The example contains two PID controllers, one controlling the level of the tank and one controlling the temperature. To increase the level, the pump is used, and to increase the temperature, the heater is used. The valve is used to decrease the level. If the valve is open and the level is kept constant, new cold water must be added. This means that the valve is used indirectly to decrease the temperature. The reference value of each controller is set by an action. The action *fill* sets the reference level to 15, while the default value is 5. The action *heat* sets the reference temperature to 30, while the default value is 20.

The main process loop is represented as an SFC, shown in Fig. 22. In the initial step, *fill* is activated and starts to fill the tank. When the tank is filled to the level 10, the step will be deactivated and the two steps in the parallel paths will be activated. The left step keeps the action *fill* active, so that the level is kept around 15. The

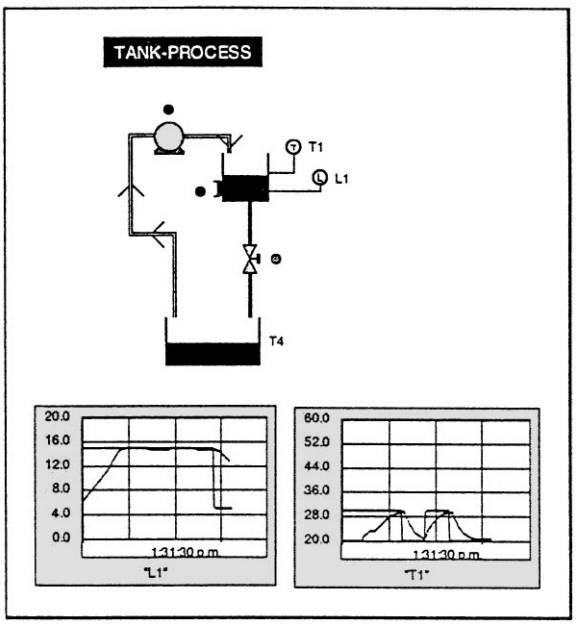

Fig. 21. The tank process.

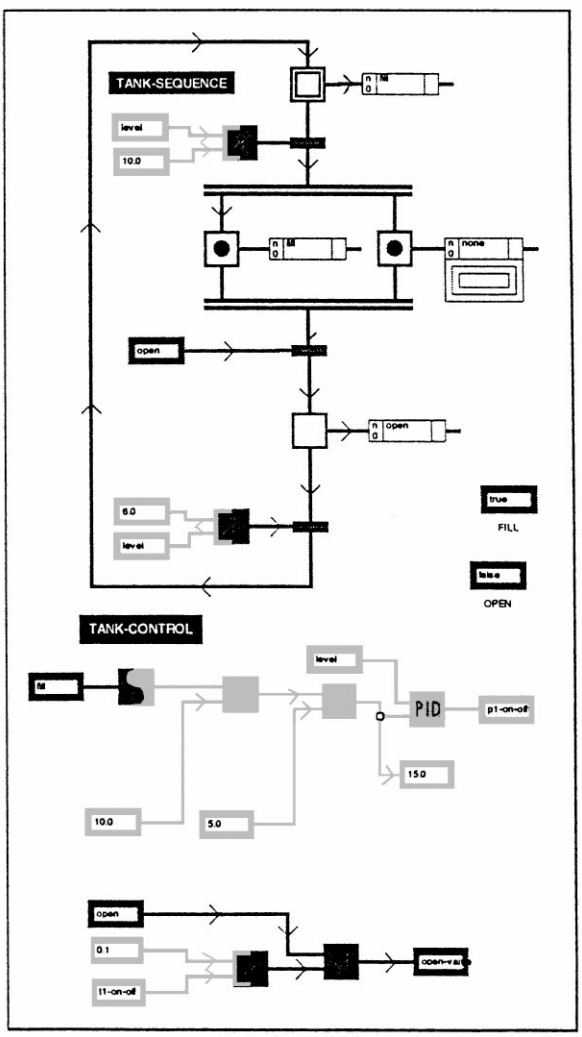

Fig. 22. The main process-control system.

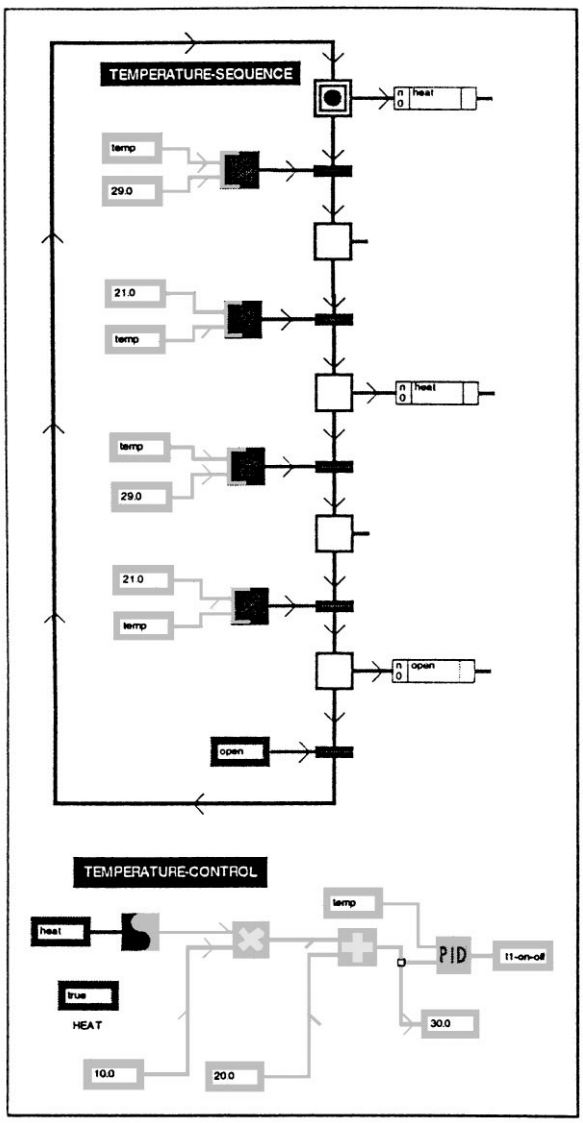

Fig. 23. The control of the temperature sequence.

step to the right is associated with an action defined on the subworkspace of the action block. This subworkspace is shown in Fig. 23.

The action contains an SFC and the PID-controller for the temperature. The SFC has two steps that are associated with the action *heat*, and two steps that are not. This gives the temperature reference signal the square wave shape of the right-hand diagram in Fig. 21. While the temperature controller sometimes opens the valve to decrease the temperature by replacing hot water with cold, the level controller must pump in more water to keep the level constant. The last step in the SFC on the subworkspace activates the action *open*, that opens the valve and activates the last step in the main SFC. This step keeps the valve open until the level of the tank has decreased to 6. Then the initial step will become active again and the pump will start to fill.

6. Conclusion

The IEC 1131-3 standard contains most of the information needed, but is sometimes not easy to understand. It is well specified how SFC and FBD function separately, but not so well how they function together. The execution order between function blocks and SFC elements is the most unclear part of the standard. Probably, the parts of the FBD connected to transitions should be executed before an SFC, and parts associated with actions after the SFC.

In the prototype implementation, all the function blocks are sorted into lists before they are executed. Algorithms for sorting the lists locally or globally are implemented. Local sorting was found to be easier to implement. However, global sorting is necessary if artificial time delays are to be avoided.

For a small example, a single procedure is written, with the code from all the function blocks in a function block diagram. The program executed an order of magnitude faster with this method. It would be interesting to implement methods for generating such procedures.

The G2 programming environment is useful as a prototype tool, since it is easy to learn and work with. Since it is slow and quite expensive, it is, however, unrealistic to use it for real PLCs.

References

David, R., Alla, H., 1992. Petri Nets and Grafcet: Tools for modelling discrete events systems. Prentice-Hall.

IEC, 1995a. Guidelines for the application and implementation of programming languages for programmable controllers. Technical Report. IEC.

IEC, 1995b IEC 1131-3. Technical Report. IEC. First edition.

Johansson, S., Öhman, M., 1995. Prototype implementation of the PLC standard IEC 1131-3. Master thesis ISRN LUTFD2/TFRT-5547-SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Lewis, R.W., 1995. Programming industrial control systems using IEC 1131-3. The Institution of Electrical Engineers.