ISSN 0280–5316
ISRN LUTFD2/TFRT--5547--SE

Prototype implementation of the PLC standard IEC 1131-3

Stefan Johansson
Martin Öhman

Department of Automatic Control
Lund Institute of Technology
December 1995

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
December 1995

Document Number
ISRN LUTFD2/TFRT--5547--SE

Author(s)
Stefan Johansson
Martin Öhman

Supervisor
Karl-Erik Årzén

Sponsoring organisation
NUTEK Regina project "High-Level Grafcet for supervisory sequential control"

Title and subtitle
Prototype implementation of the PLC standard IEC 1131-3

Abstract

IEC 1131-3 is a standard for programming languages used in programmable controllers (PLC). It consists of four language paradigms, two graphical and two text based, and sequential function charts (SFC).

This master thesis describes a prototype implementation of the Function Block Diagram language combined with SFC. Different function blocks and SFC elements are implemented as a number of classes and code is written to execute them.

We have spent most of the project on issues of execution order which are not well specified in the standard. The objects are sorted into lists and executed. Algoritms for local or global sort are implemented and evaluated. We have also discussed other execution methods.

The implementation is done using the graphical and object oriented expert system G2 as a prototype tool.

Key words
G2, IEC 1131, PLC

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
50

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Fax +46 46 110019, Telex: 33248 lubbis lund.

Acknowledgements

We would like to thank our supervisor *Karl-Erik Årzén* for giving us good guidance and support. A special thanks for arranging meetings with the industry where we got a lot of inspiration for our work.

We would also like to to thank *Leif Andersson* for helping us with LaTeX.

Thanks also to our opponents *Ulf Holmgren* and *Michael Templin* for giving constructive feedback on our work.

Finally we would like to thank all the staff here at the Department of Automatic Control for making our time here very nice.

Lund, December 1995

Stefan Johansson Martin Öhman

Contents

1. Introduction 5

2. Programmable Logic Controllers 6

3. The standard IEC 1131-3 7

3.1 Programming languages 7

3.2 Program Organization Units 8

3.3 Configuration Elements 10

3.4 Communication 11

3.5 Data types 11

3.6 Sequential Function Charts 13

3.7 Unclear parts 16

4. General implementation aspects 19

4.1 Function blocks 19

4.2 Sequential Function Charts 20

4.3 Execution 21

4.4 Tasks 26

4.5 Future work 28

5. Implementation using G2 29

5.1 Function blocks 30

5.2 Hierarchical function blocks 33

5.3 Sequential Function Charts 34

5.4 Connections 34

5.5 Global variables 36

5.6 Execution 36

5.7 Tasks 36

6. An example 38

7. Conclusions 42

8. Bibliography 43

A. G2 44

A.1 Classes and objects 44

A.2 Relating objects 45

A.3 The inference engine 46
A.4 Procedures 47
A.5 Simulation 47
A.6 Development interface 48
A.7 End-user Interface 48
A.8 External interfaces 48
A.9 Drawbacks 49

B. Implementation files 50

1. Introduction

IEC 1131-3 is a standard for programming languages used in programmable controllers (PLC). It was defined in March 1993 by the International Electrotechnical Commission (IEC). The standard will probably be commonly used in the future.

IEC 1131-3 consists of four language paradigms, two graphical and two text based, and Sequential Function Charts for internal structuring of the programs. The purpose of this master thesis is to make a prototype implementation of Function Block Diagrams (FBD), one of the graphical languages, and focus on how it interacts with the Sequential Function Charts (SFC).

The experience from the thesis could be used in future large scale commercial projects implementing PLC programming languages according to the standard. The goal of the work is to:

- Test different implementation methods and detect the problems that may occur when implementing them.
- Point at unclear parts, unspecified parts and errors in the standard.
- Describe the FBD and SFC parts of the standard.
- Find out to what degree G2 is a suitable tool for implementing the standard.

We have used the real-time expert system G2 from Gensym Corporation. G2 is graphical and object oriented. We chose G2 because it is easy to learn and very fast to work with.

Chapter 2 contains a brief description of what a PLC is. In Chapter 3 the FBD and SFC parts of the standard are described. In Chapter 4 the general implementation ideas that could be used in any programming environment are explained. Chapter 5 contains the experience of implementing the standard in G2. An example of how to use FBD combined with SFC is given in Chapter 6. In Chapter 7 the conclusions are summarized. Appendix A is a short introduction to G2 and Appendix B describes the program files.

2. Programmable Logic Controllers

PLCs (Programmable Logic Controllers) are specialized computers that are widely used in industrial automation. The first PLCs were designed in the mid 70's to replace the old relay control systems. They consisted of a single bit processor with a program memory, an accumulator and a number of outputs and inputs. These early PLCs could only handle digital input and output signals and perform simple logical functions.

Today the PLCs are more advanced and can handle analog signals as well as doing advanced calculations like PID control. There are many different sizes of PLCs and the larger ones are often programmed from a workstation.

The execution of a PLC program follows three phases. First all the input signals are read into the PLC. Then the PLC program is executed once. Finally the results are written to the outputs. These phases are repeated in an infinite loop. This can be seen in Figure 2.1.

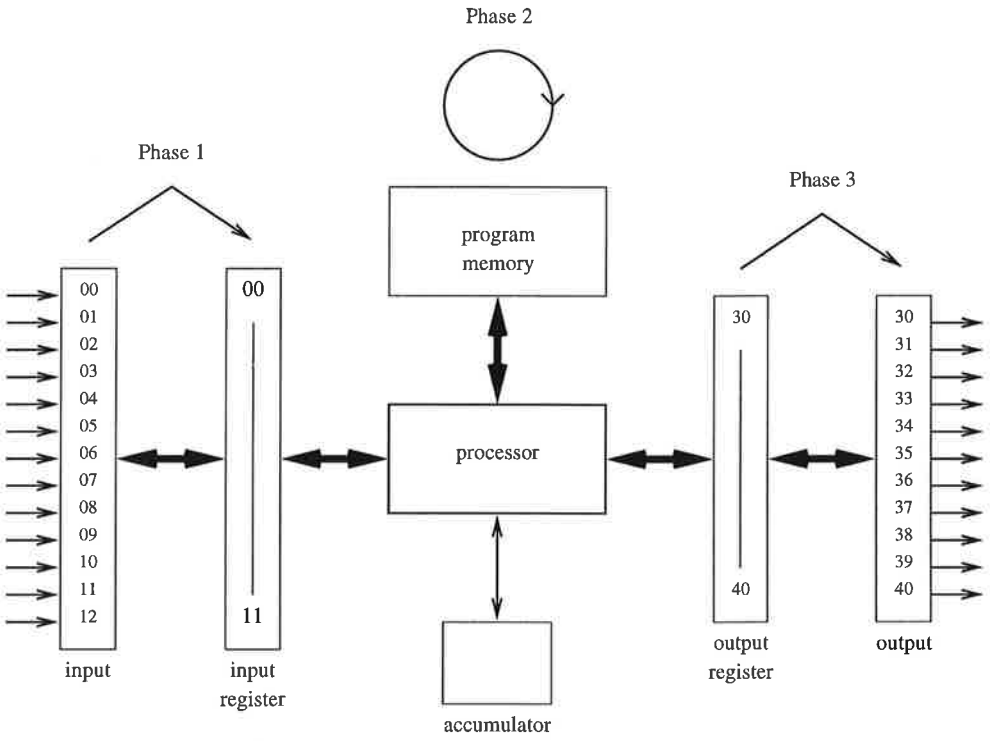


Figure 2.1 A schematic picture of a PLC.

3. The standard IEC 1131-3

The International Electrotechnical Commission (IEC) has defined a standard for programmable controllers, IEC 1131. The standard consists of the following parts:

1. General information.
2. Equipment requirements and tests.
3. Programming languages.
4. User guidelines.
5. Messaging service specification.

This thesis concerns part 3, the programming languages.

3.1. Programming languages

There are four language paradigms and one method for their internal structuring defined in IEC 1131-3. The standard specifies different program organization units and their representation, rules of evaluation and communication mechanisms. The standard also describes which standard functions should be implemented.

Languages

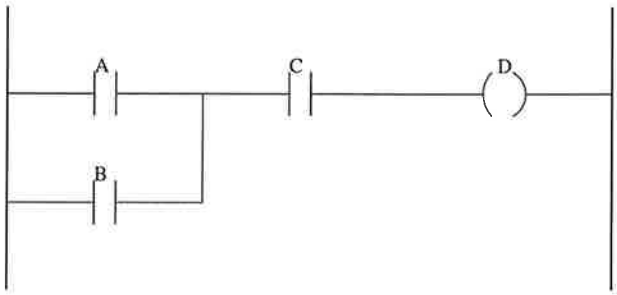
Instruction List The instruction list language is similar to assembly code with commands like load and store. This language corresponds to the programming technique that has traditionally been used with PLCs. It uses an accumulator to store results.

```
FUNCTION D : BOOLEAN

  VAR_INPUT
    A, B, C : BOOLEAN;
  END_VAR

  LD A          (* Load A and store it in the accumulator   *)
  OR B          (* Or B with the accumulator and store the  *)
                (* result in the accumulator                *)
  AND C         (* And C with the accumulator and store the *)
                (* result in the accumulator                *)
  ST D          (* Store the accumulator value in D         *)

END_FUNCTION
```

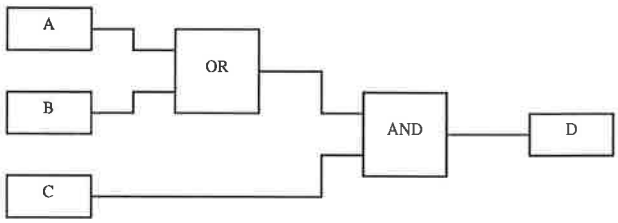
Structured Text The structured text language is similar to pascal. It has sequential statements, conditional statements like IF and CASE and repetitive statements like FOR ... DO ... END_FOR, WHILE ... DO and REPEAT ... UNTIL.

```
FUNCTION D : BOOLEAN

  VAR_INPUT
    A, B, C : BOOLEAN;
  END_VAR

  D := A OR B AND C;

END_FUNCTION
```

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



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



3.2. Program Organization Units

When programming a PLC according to IEC 1131-3, three so called program organization units are used: functions, function blocks and programs. The program organization units can be defined by the user or by the manufacturer. They can not be recursive. This means that a unit can not contain a call to itself. Figure 3.1 shows how a program is defined using the different program organization units.

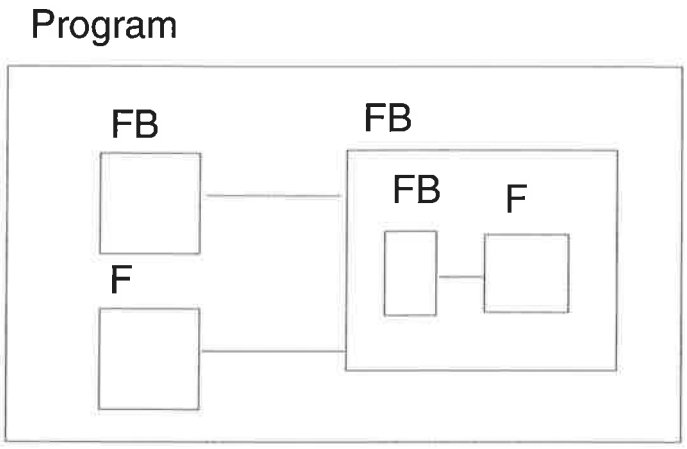


Figure 3.1 The relation between program organization units

Functions

Functions have one or more inputs but only one output. To execute a function all inputs must have a value. A function can not store any state information. That means that executing the function with certain values on the inputs always gives the same output value.

Functions can be hierarchically defined, by using already defined functions when building up new ones.

A number of standard functions shall be supplied by the manufacturer. Some of the functions shall be *extensible*, meaning that the user can decide the number of inputs to the function. The functions are either *typed* or *overloaded*. Typed functions operate on a specific data type, while the overloaded ones can operate on different types.

The standard functions include for example numerical functions like *add* and *sine* and logical functions like *and* and *or*.

Function blocks

Function blocks have one or more inputs and also one or more outputs. To execute a function block all inputs must have a value. A function block can store state information. This means that the values of the outputs depend not only on the values of the inputs but also on the state information.

The user must give each instance of a function block an individual name. Function blocks can be hierarchically defined, by using already defined functions and function blocks. A PID controller could be an example of a user defined function block.

There shall also be a number of standard function blocks supplied by the manufacturer. Counters and flip-flops are examples of standard function blocks. Standard function blocks for communication are defined in IEC 1131-5.

Programs

A program is built up by functions and function blocks. Programs do not have inputs or outputs.

3.3. Configuration Elements

Besides the program organization units there is a number of so called configuration elements that support the installation of programs into programmable controller systems. They are tasks, resources, configurations, global variables and access paths. Figure 3.2 shows how they function together.

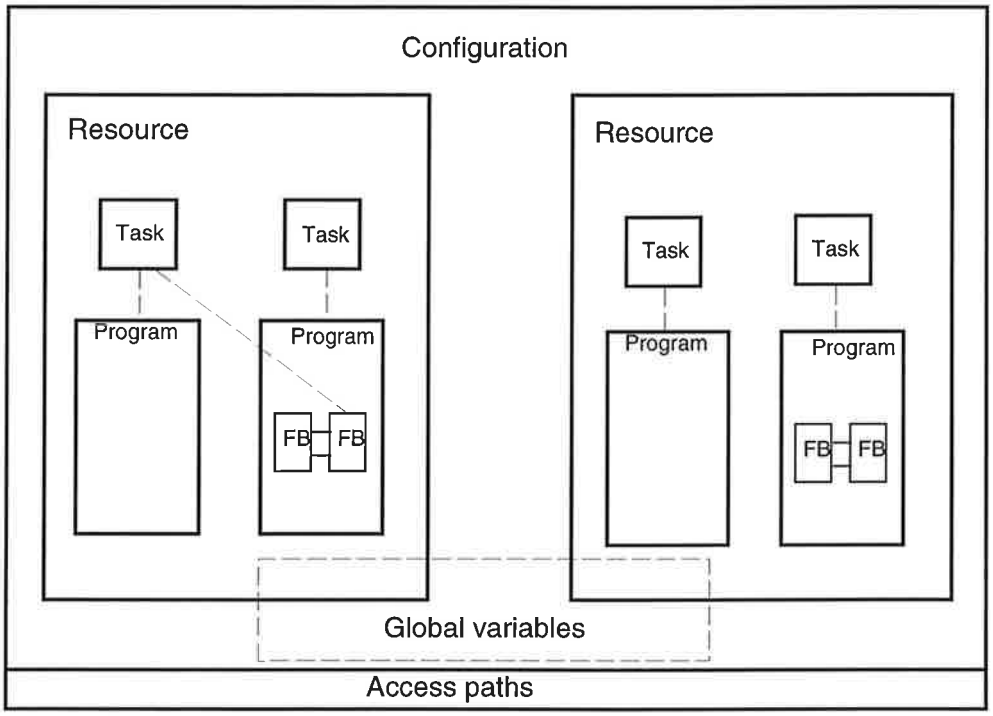


Figure 3.2 The configuration elements

Tasks

A task controls the execution of programs and function blocks. There are three inputs to a task: *single*, *interval* and *priority*. A task 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 *priority*.

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. See Figure 3.3.

Resources

Any signal processing function with its man-machine interface, sensor and actuator interface is called a resource. In most cases this means a processor

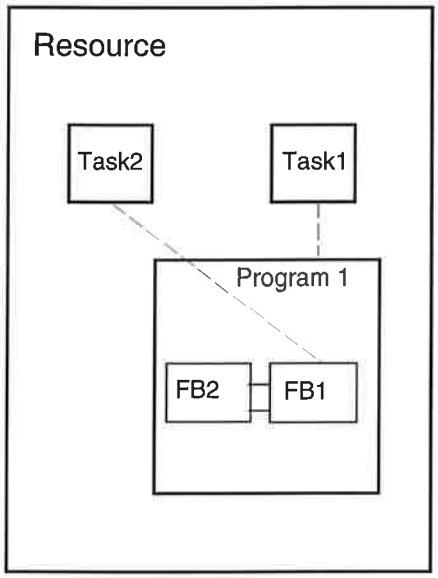


Figure 3.3 A function block is indirectly controlled by two tasks

with an I/O unit. A resource contains programs and tasks.

Configurations

Each programmable logic controller is called a configuration. A configuration contains one or more resources.

Global variables

Global variables can be used in programs, resources and configurations.

Access paths

Access paths are needed to use the communication services specified in IEC 1131-5 to communicate with systems outside the configuration.

3.4. Communication

There are different ways to transfer values. Figure 3.4 shows that there can be data flow between functions or function blocks within a program. Data can also written to a global variable. Other programs can then use the data. This is shown in Figure 3.5. To communicate between configurations, send and receive function blocks can be used. This is shown in Figure 3.6. These blocks are described in IEC 1131-5.

3.5. Data types

The standard specifies a number of data types like boolean, integer and real. The user can also define new data types. These can be simple types or structured types.

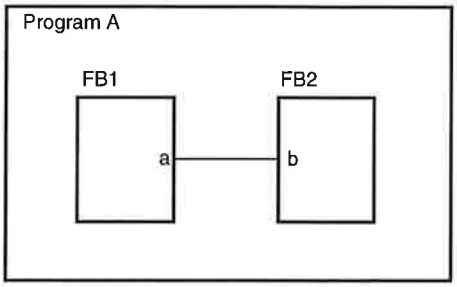


Figure 3.4 Communication between function blocks

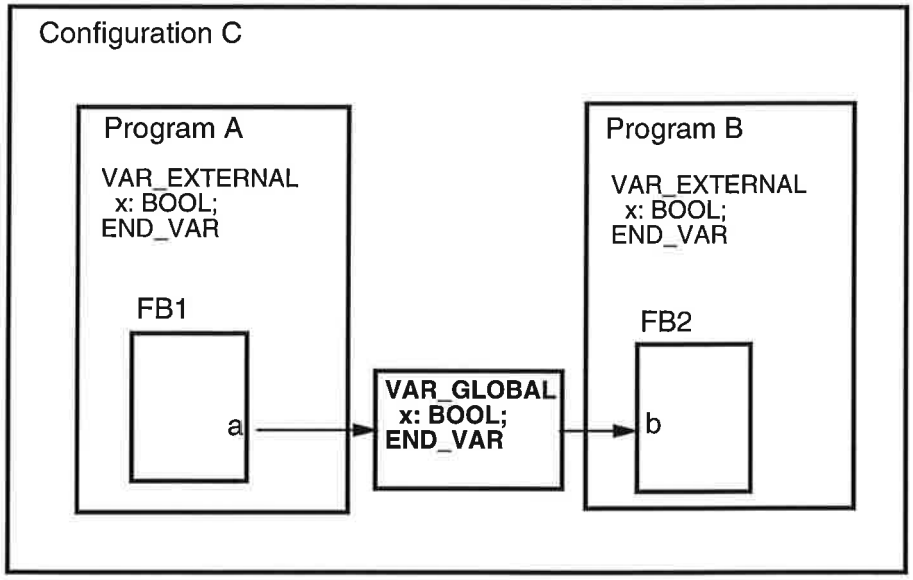


Figure 3.5 Communication between programs

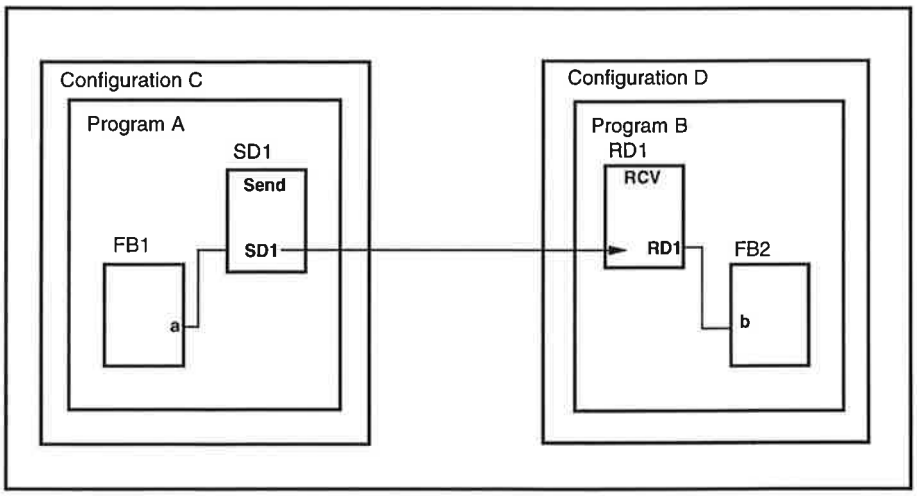


Figure 3.6 Communication between configurations

3.6. Sequential Function Charts

A Sequential Function Chart (SFC) is an extended state machine, also known as a grafcet. An SFC is a directed graph constisting of two main elements, *steps* and *transitions*.

Grafcet was developed as a French standard in the 70's. In the 80's this was included in the standard IEC 848 were the name was changed to SFC. The SFC part of IEC 848 has been included in IEC 1131-3.

According to the standard, SFC is a way of structuring the program organization units programs and function blocks implemented in any of the four languages. If any part of a program organization unit is organized with SFC the whole unit must be so. A unit that is not structured is said to be a single action (see below) "controlled by the invoking entity". This means that the program organization unit is executed continuously.

Step

A step is represented graphically by a rectangle. Steps can be active or inactive. The state of the SFC is determined by which steps that are active and inactive, respectively. Steps are initially inactive. Special *initial steps* are initially active. A step and an initial step are shown in Figure 3.7.

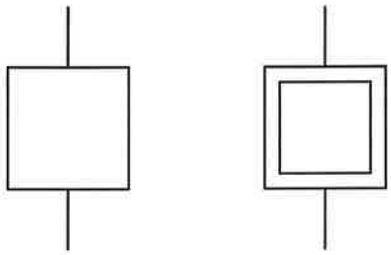


Figure 3.7 A step and an initial step

Transition

A transition connects two steps with each other, see Figure 3.8. A transition has a *transition condition* that is a boolean input and can be true or false. The transition condition can be described by any of the four languages or by a boolean variable. The transition condition can not be evaluated until it has got a value.

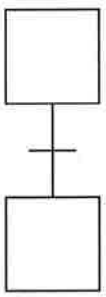


Figure 3.8 A transition between two steps

A transition will fire when the step above it is true and the transition condition is true. When the transition fires the step above it will be deactivated and the step below it will be activated. This can be seen as a token that is moved from one step to the other.

Parallel and alternative paths

Two or more transitions can follow a step. This is called *alternative paths*. If any of the transition conditions is true and the step above them is active that transition will fire.

Two or more steps can follow a transition. This is called *parallel paths*. When the transition fires all the steps below will be activated. When the parallel paths join together again a single transition will follow a number of steps. All the steps preceding the transition must be active in order for the transition to take place. Figure 3.9 shows alternative and parallel paths.

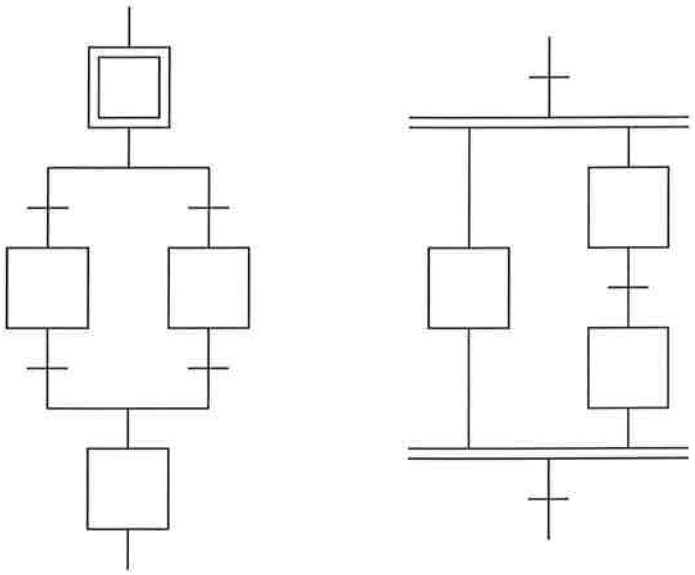


Figure 3.9 An alternative path and a parallel path

When working with parallel paths the user must avoid building illegal SFCs. An SFC can be illegal in two ways: *unsafe* or *unreachable*.

In Figure 3.10 S2 can stay active while S3, S5, S7, S1 become active one after another. The S2 and S3 will be activated while S2 still is active. This is called an unsafe SFC.

An SFC is unreachable if there can be a deadlock situation where no steps can activated or deactivated because of the states of other steps. Figure 3.11 shows an unreachable SFC.

Actions

It is possible to associate zero or more *actions* with a step. An action can be described with any of the four programming languages, be described as an SFC or be a single boolean variable.

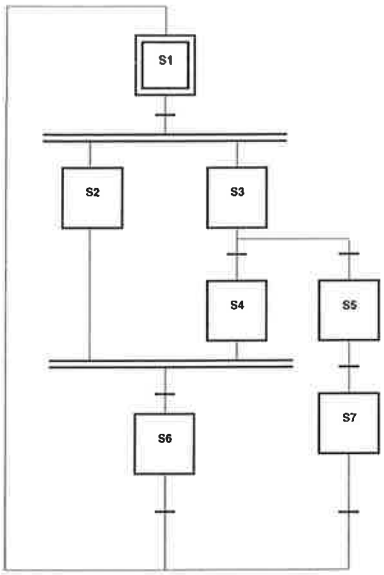


Figure 3.10 An unsafe SFC.

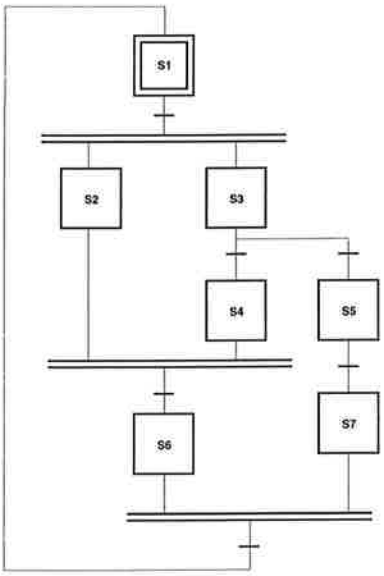


Figure 3.11 An unreachable SFC.

Action blocks

To associate actions with steps *action blocks* are used. An action block has a boolean input that indicates when the associated step is active and a boolean output which the standard does not specify the use of. The action block can therefore function also as a normal boolean function block with one input and one output. To associate a boolean action with an action block the name of the action is written in the action block. A non-boolean action can be associated with only one action block and is therefore declared directly in the action block. Steps, actions and action blocks relate in the following way:

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

Figure 3.12 shows that the boolean action A1 is associated with the action blocks AB1 and AB2. The non-boolean action A2 is constructed directly in the action block AB3 and associated only with that action block. All actions, boolean and non-boolean, must have a unique name.

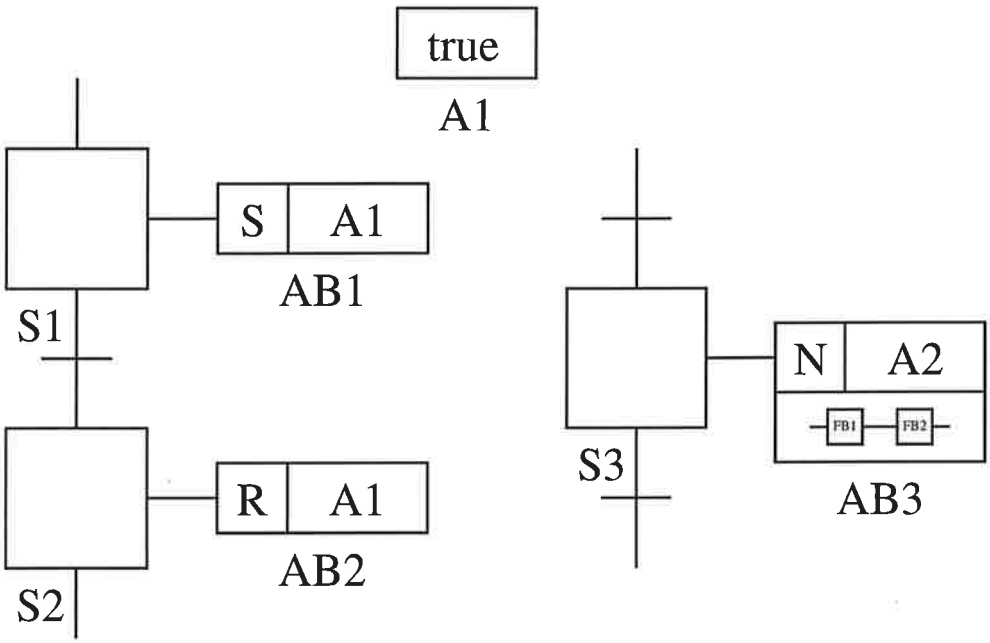


Figure 3.12 Action blocks associated with boolean and non-boolean actions.

Action qualifier

An action block uses an *action qualifier* to control the action. The action will notice which action blocks on the associated action blocks that are active. Then it will execute depending on these qualifiers. There are a number of different action qualifiers. They are shown in Table 3.1.

Action blocks associated with the same action are not allowed to be active at the same time if they have certain combinations of action qualifiers. See Section 3.7.

3.7. Unclear parts

The standard is quite "heavy" and some parts are hard to read. Many issues are not specified or unclearly specified, especially concerning execution order. A couple of things are probably not meant the way they are written.

Qualifier	Symbol	Explanation
null	None	The action is never activated.
Non-stored	N	The action is active while the action block is active.
Stored	S	The action is activated when the action block becomes active.
Reset	R	The action is deactivated when the action block becomes active.
Limited	L	The action is activated when the action block becomes active and is deactivated after a certain time or when the action block becomes inactive.
Delayed	D	The action is activated a certain time after the action block becomes active and is active as long as the action block is active
Pulse	P	The action is active for one clock cycle when the action block is activated.
Stored and delayed	SD	The action is activated a certain time after the action block becomes active.
Delayed and stored	DS	The action is activated a certain time after the action block becomes active if the action block is still active.
Stored and limited	SL	The action is activated when the action block becomes active and is deactivated after a certain time.

Table 3.1 Action qualifiers

The standard specifies that there can be multiple instances of function blocks. We believe that it should be specified that also functions can have multiple instances.

If a standard function is extensible it means that it can have a variable number of inputs. The standard does not specify if the user has to decide the number of inputs when he creates the function or if the number can be changed during the lifetime of the function.

According to the standard it shall be possible to do jumps in all the four languages. We think this is not so good to use in the FBD language. [2] also advices that this is not used.

It is said in the standard that a function can be defined from other functions and that function blocks can be defined from other functions and function blocks. It is not specified if such a function or function block should be

considered a fixed unit or if it has an internal structure that can be studied and changed.

If functions and function blocks can be considered to have a hierarchical structure, it is not specified if the implementation should use local or global sorting. See Section 4.3 for an explanation of these terms.

The execution order between functions and function blocks, SFC-elements, action blocks and actions is not specified. It is written in the standard that no element shall be evaluated until the states of all its inputs have been evaluated. We believe that this leads to the execution order shown in Figure 3.13:

1. Functions and function blocks connected to an input of a transition.
2. All SFC elements.
3. Action blocks, function blocks connected to them and non-boolean actions associated with them.
4. Boolean actions.

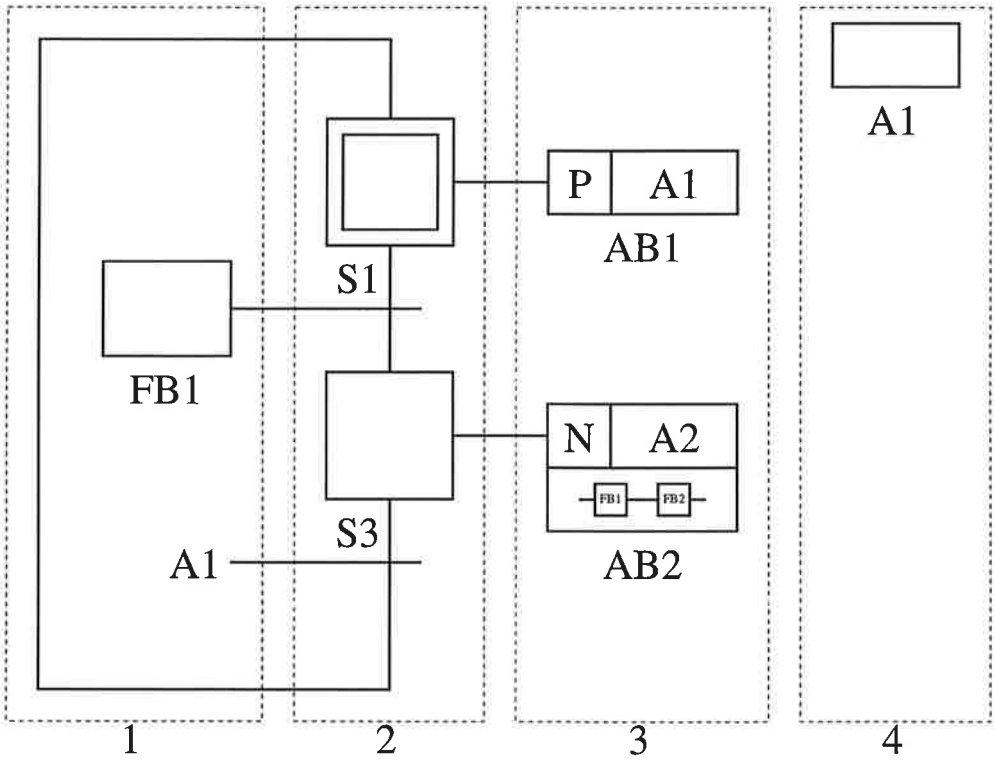


Figure 3.13 The execution order of functions, function blocks, steps, transitions, action blocks and action

4. General implementation aspects

This chapter describes general implementation aspects that are environment independent. We assume, however, an object oriented programming environment. Chapter 5 describes the parts of the implementation that are specific for G2.

There are classes for all function blocks and SFC objects. All graphical objects on the workspaces are instances of these classes.

Our implementation only supports one resource. The resource is represented as a workspace (a window). The user constructs the function block diagram (the programs) on the workspace with graphical objects.

4.1. Function blocks

Since programs and functions can be seen as special cases of function blocks we have only implemented *function blocks* that the user can use as functions, function blocks or programs.

Our implementation contains function blocks with zero to two inputs and zero or one output. The user uses zero-input zero-output function blocks as programs and single output function blocks without internal state information as functions. We have implemented input and output blocks using zero-input one-output respective one-input zero-output function blocks. Figure 4.1 shows a function block diagram.

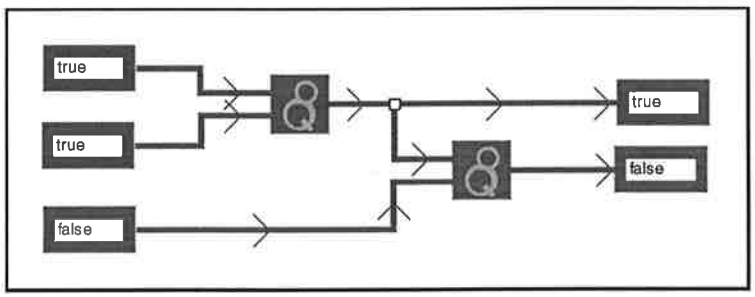


Figure 4.1 A function block diagram

Function blocks are represented graphically as rectangles with a symbol inside indicating the purpose of the function block. The connections between the function blocks are represented as lines with arrows indicating the data flow direction and colors indicating the data type.

All hierarchical function blocks have their own subworkspace where their internal structure is defined. It is necessary to have a link from the function block to its workspace. Figure 4.2 shows a hierarchical function block diagram and its subworkspace.

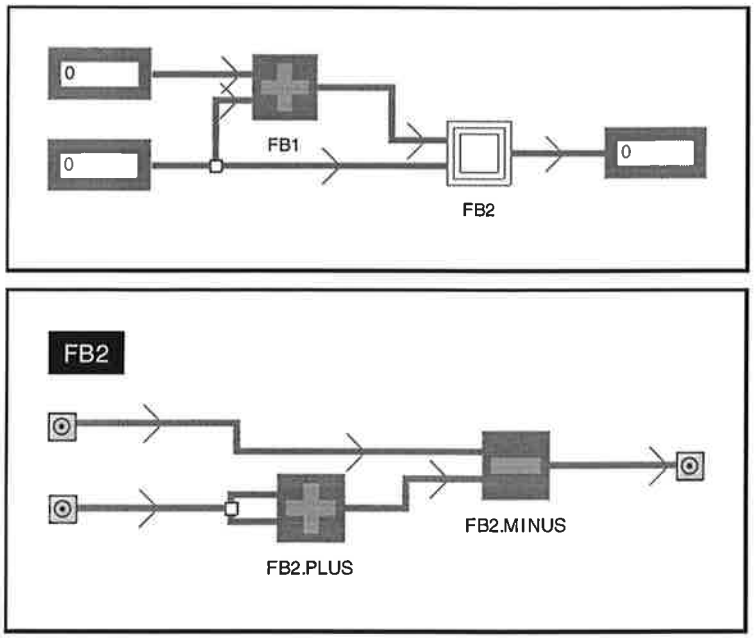


Figure 4.2 A hierarchical function block diagram

4.2. Sequential Function Charts

The main classes in our implementation of SFC are steps, transitions, actions and action blocks.

All steps and transitions have an input and an output, used for connecting them with each other. Steps are represented graphically as rectangles. They have filled red circles indicating when they are active. This is also written to a boolean output that can be connected to an action block. Initial steps are a subclass of steps and are initiated in an active state.

Transitions are represented graphically as a horizontal lines. They have a boolean input that acts as a transition condition.

We have implemented special classes for path branches and joins. They are alternative divergence, alternative convergence, parallel divergence and parallel convergence.

Steps, transitions, branches and joins all have methods that are used to determine whether a step should be activated or deactivated. They often call each other.

Action blocks are implemented similarly to function blocks. Action blocks associated with actions that are defined with function blocks or SFCs are implemented similarly to hierarchical function blocks. The action is then constructed on the subworkspace of the action block. Action blocks have a boolean input and a boolean output and can also function as normal function blocks.

Actions are a subclass to boolean variables. If the action only function 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 can only be associated with one action block. The action is then an attribute of that action block and has no graphical representation. The standard describes the functionality of an action with a function block diagram. We have implemented this functionality in a method of the action class.

Figure 4.3 shows an SFC with steps, transitions, action blocks and actions. The boolean action A2 is associated with the action blocks AB1 and AB3. The non-boolean action A1 is defined on the subworkspace of the action block AB2.

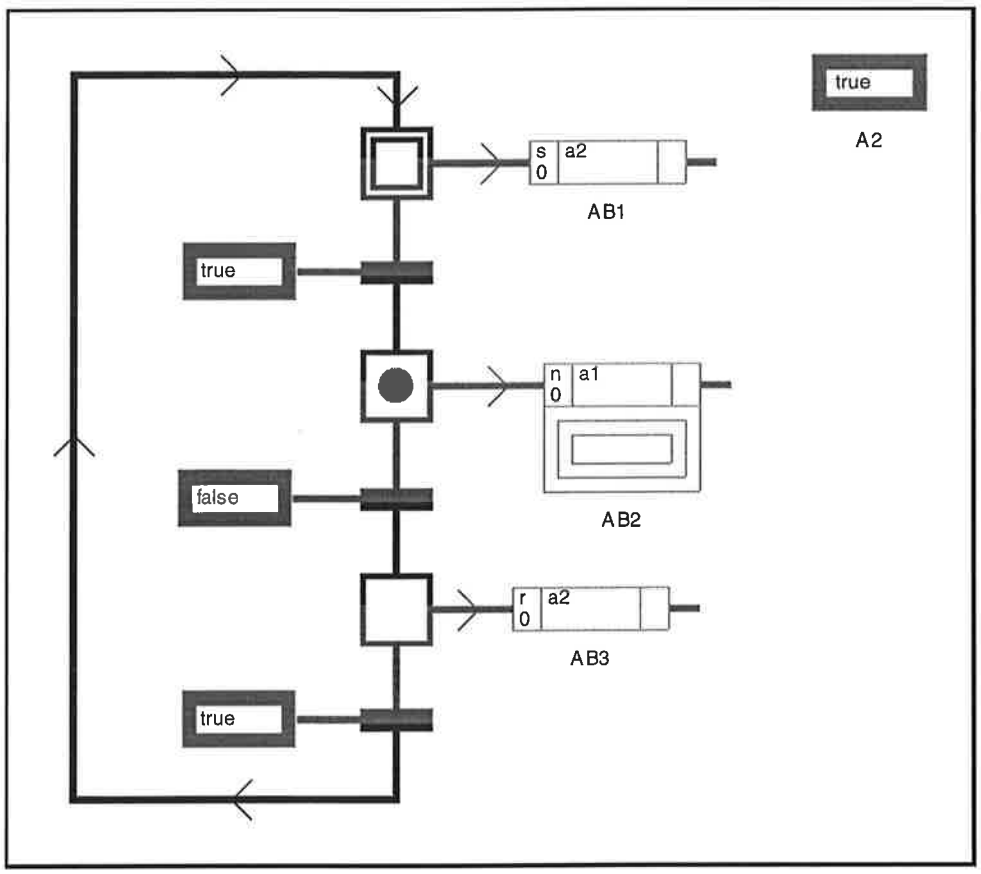


Figure 4.3 An SFC with steps, transitions, action blocks and actions

4.3. Execution

It is not specified in the standard if SFC elements or function blocks should be executed first. In the implementation steps are executed first and then function blocks. We believe however that this is not the most correct order of execution. See Section 3.7. Action blocks and actions are executed together with the function blocks.

The order in which the function blocks are executed must follow specific rules. A function block can not be executed until the function blocks on all its inputs have been executed.

This rule causes problems when building a loop. In Figure 4.4 FB1 must be executed before FB2 to give the input of FB2 a value. For the same

reason, however, FB2 must be executed before FB1. This problem is called an *algebraic loop*.

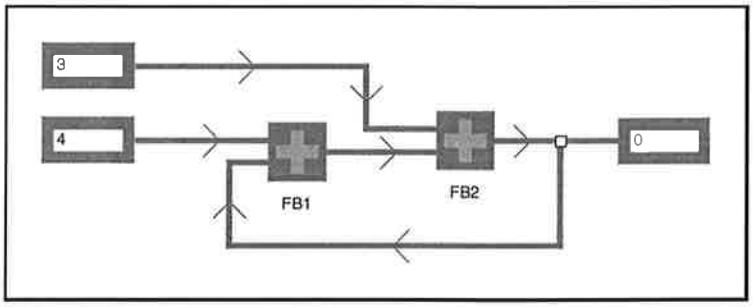


Figure 4.4 Algebraic loop

This problem is solved by introducing so called *loop elements*. It is up to the user to decide where to break the loop and insert a loop element. The only property of the loop element is that it can be executed before the function block that is connected to its input. It reads the old value from the input and writes it to the output. Since the loop element uses the old value, the user can use them, instead of variables, as a memory. In Figure 4.5 we have solved the problem with the algebraic loop by inserting a loop element. FB1 can now be executed before FB2.

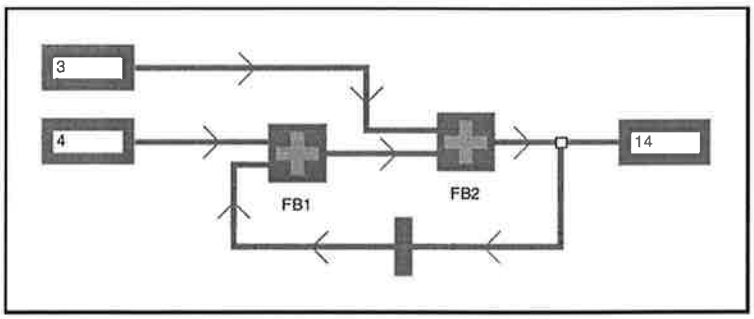


Figure 4.5 Loop element

The standard specifies that all steps should be updated synchronously. This problem is solved by going through all the steps twice. Figure 4.6 shows a small example of how an SFC is executed. In the first phase it is checked which steps can be activated and they are marked with the flag "next". In the example only the middle step can be activated. In the second phase all marked steps are activated and the steps above them are deactivated. This method makes sure that each token can pass only one transition at a time. If the steps would be checked and activated in the same phase a step could be first activated and then deactivated in the same task cycle. In Figure 4.6 the middle step could first be activated and then could the bottom step be activated and deactivate the middle step. This is against the rules of execution of SFC. In this solution with two phases the order in which the steps are treated within a phase is not important.

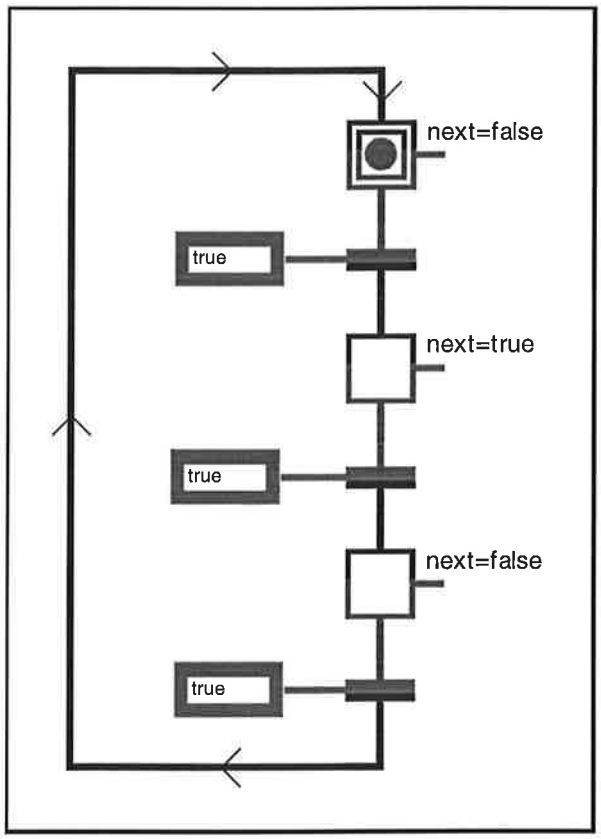


Figure 4.6 Execution of an SFC

Execution list

To make sure that the rules of execution are followed, a function block list is built in which all the function blocks are inserted in the correct execution order. A step list is also built where all steps are inserted at random order. The lists are built when the task is initialized.

Each time a user function block is executed it will use the lists. First it will go through the step list twice while doing the two phase algorithm of execution of an SFC described above. Then all function blocks in the function block list are executed.

Loop elements are placed first in the function block list and thereafter function blocks without inputs. Then function blocks are inserted which have all their inputs connected to function blocks already in the list. This is repeated until all function blocks are in the list. Hierarchical function blocks can be dealt with in two ways: local and global lists.

Local lists One method is to let each hierarchical function block have its own lists containing all the function blocks and steps on its workspace.

Figure 4.2 shows a hierarchical function block diagram. In Figure 4.7 the main function block list only contains the function blocks on the top level. A local list of FB2 contains the function blocks on the next level. Steps are inserted in step lists in a corresponding way.

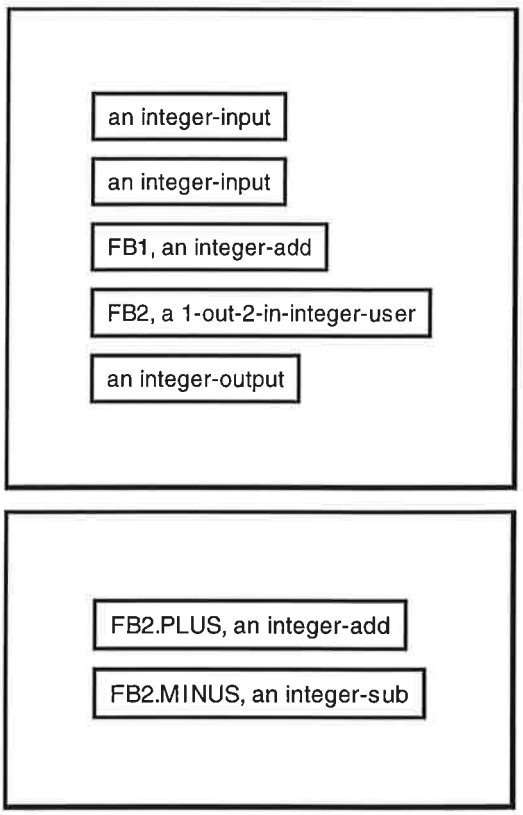


Figure 4.7 The execution list of the function block diagram in Figure 4.2 and the local list of FB2.

The main advantage with local lists is that if you make changes in a hierarchical function block, you only have to build a new list for that function block, not for the whole function block diagram. This corresponds to separate compiling of program units.

Global lists The other method is to have only one function block list and one step list for each top level function block containing all function blocks and steps on the workspaces of function blocks at all levels. No hierarchical function blocks are inserted in the list, only the function blocks and steps on their workspaces.

Figure 4.8 shows the global list of the function block diagram in Figure 4.2. Note that FB2 is not in the list.

The main advantage with a global list is that you can get rid of imaginary loops that seem to exist from a local point of view. If the function block diagram in Figure 4.9 would be sorted into local lists the user would have to insert a loop element. The global list shows that this is not needed with global sort. Removing loops gives you a faster implementation by saving clock cycles.

A disadvantage with global lists is that the implementation is more difficult. When a hierarchical function block is disabled it is supposed to disable all function blocks on its subworkspace. If they are spread out in a global list this will be difficult. When working together with SFCs it is

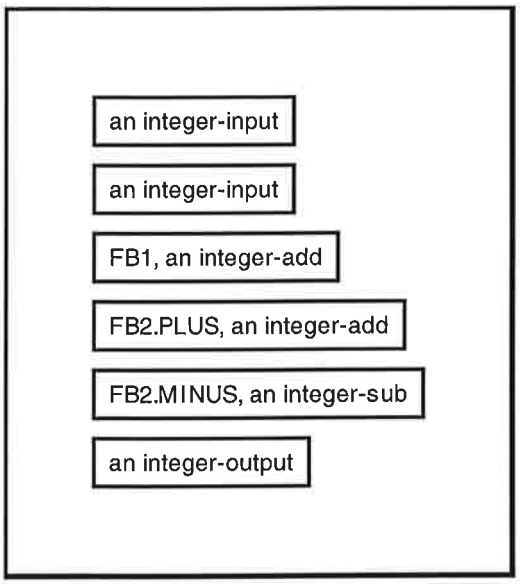


Figure 4.8 The global execution list of the function block diagram in Figure 4.2

important that function blocks can be disabled because this is used by the actions to control the function blocks.

When a function block diagram or an SFC is built on a subworkspace of an action block, the action block functions like a hierarchical function block with an enable signal associated with an action. The function blocks and steps on the subworkspace are placed in local lists even when the rest of the resource is sorted globally. Then the execution of these items can be controlled by the action.

Execution procedure

The execution problem can be solved without using lists. One solution is to replace the lists with a procedure. This procedure would be generated automatically when the task is activated. It would contain the code from all the function blocks inserted in the same order as in the list. Each time the task is executed it would only execute the procedure. We have not implemented this, but we have made a small example to evaluate the time differences. The example consists of five inverter functions, see Figure 4.10.

We have compared the execution time between an execution list, Figure 4.11, and a procedure, Figure 4.12. They were both executed 100 000 times. The list took 112 s and the procedure took 7 s. This shows that there is a lot to gain with this method.

Asynchronous dataflow

Another solution would be to let the execution be asynchronously controlled by the data flow. Each function block would then be executed as soon as the value of any of its inputs is changed. This does not correspond well to the standard. The standard specifies that a task should execute all its associated function blocks each task cycle. This should not be dependent of if the inputs of the function blocks have received new values or not.

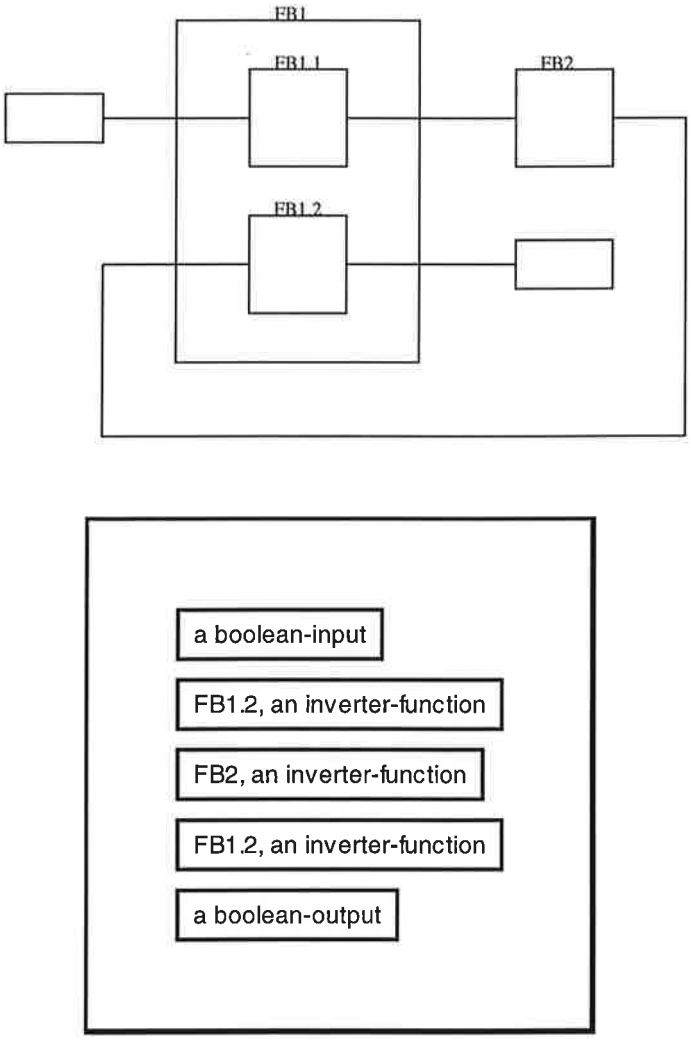


Figure 4.9 A loop is removed by using global lists.

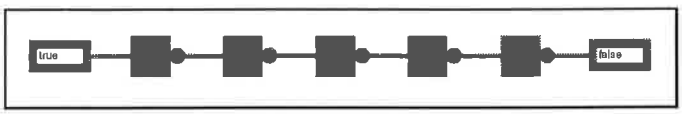


Figure 4.10 The timing example.

4.4. Tasks

To control the different tasks we have defined a *task handler*. Each time unit the task handler calls every task. Therefore there must be a connection between the task handler and the tasks.

Every task has a counter. When the task handler calls a task, the counter is decreased. When the counter reaches 0 the task will execute and then set its counter to the execution interval.

When a task executes it runs all its associated function blocks. Hence there need to be links between tasks and their associated function blocks. We

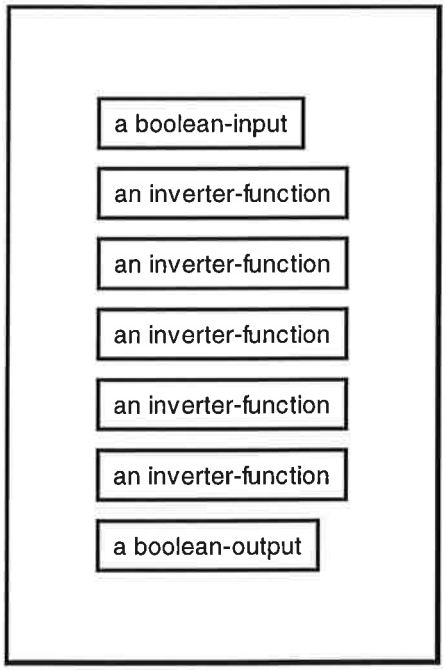


Figure 4.11 The execution list of the timing example.

```
more-time ()=()
n: integer;
x: truth-value;
begin
    inform the operator that "time = [the current real time] ";
    for n = 1 to 100000 do
        x = true;
        x = not (x);
        x = not (x);
        x = not (x);
        x = not (x);
        x = not (x);
    end;
    inform the operator that "time = [the current real time] ";
end
```

Figure 4.12 The procedure of the timing example.

let each function block store the name of its task. When a task is activated it builds up a dynamic list with all its function blocks (see Chapter 5). Another solution would be to let the task have a static list with pointers to all its function blocks.

A function block can indirectly be controlled by more than one task. In Figure 3.3 Task1 executes FB1 and Task2 executes FB1.1 directly. If they try to execute at the same time and if preemptive scheduling is used, there might be problems.

4.5. Future work

This section lists the points where our implementation does not follow the standard. We also mention some of the features in the standard that we have not implemented.

The standard specifies that it shall be possible for a configuration to contain multiple resources. We have worked on a single workstation and our implementation only supports one resource.

We have worked with G2 as a closed environment. IEC 1131-5 describes how *access paths* are used to communicate outside the configuration. We have not studied this.

According to the standard, a program that is not associated with a task shall execute continuously but with the lowest priority. In our implementation only programs associated with a task execute. We have not implemented support for tasks to have priorities.

It should be possible for the user to derive his own data types. We have not implemented this. We have also not implemented all the standard types and standard functions specified in the standard. This since our implementation is only to be seen as a prototype and not as a complete implementation of the standard.

An important issue in the standard that we have not had time to implement is the ability to enable and disable function blocks. This would be good to implement so that actions that are global variables can control function blocks.

Some standard functions are supposed to be overloaded and/or extensible. We have not implemented these features.

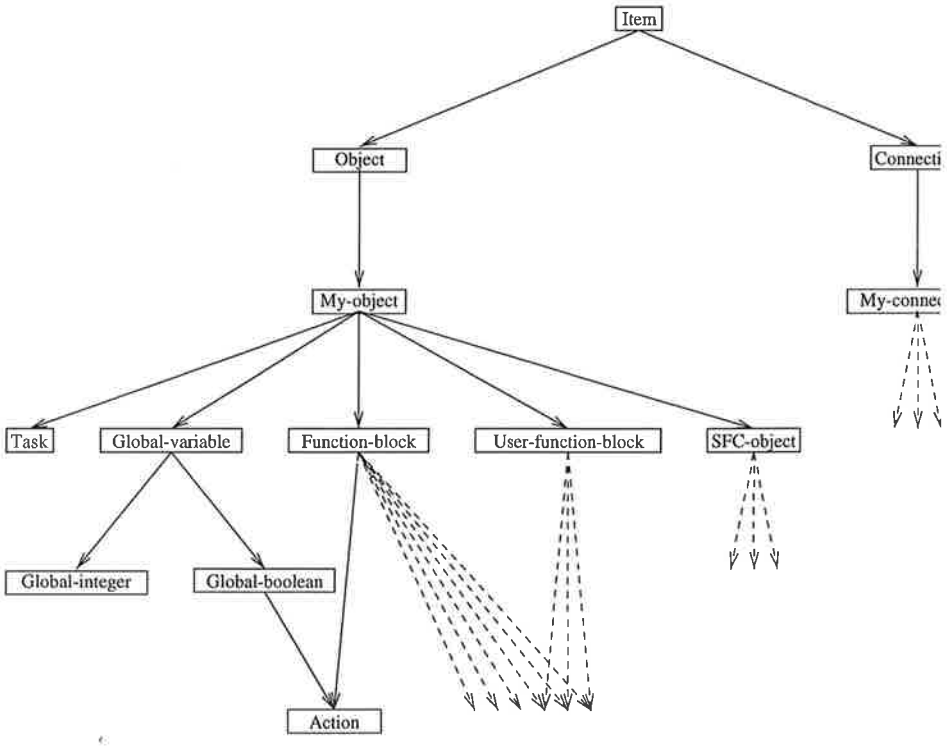


Figure 5.2 The main class hierarchy.

5.1. Function blocks

Function blocks are placed in the class tree according to their number of inputs and outputs. Each group of function blocks with a specific number of inputs and outputs belongs to a common class. These classes inherit from two classes, one giving the number of inputs and one the number of outputs. The combinations of inputs and outputs gives a matrix with possible groups of classes. This matrix is shown in Figure 5.3. We would like to extend the matrix with one more dimension representing the data type of the function block. However, since in G2 it is not possible to inherit the color of the connection stubs, we found no advantage of having this third dimension. Each group of classes is described in the following subsections. For each group, examples of the implemented classes are given.

0-out-0-in

The class 0-out-0-in-user is a hierarchical function block without inputs and outputs. This is used by the user as a *program*. An action is also a 0-out-0-in-function block. See subsection 5.5. Figure 5.4 shows the 0-out-0-in class hierarchy.

0-out-1-in

Classes with input but no output include *output* function blocks which are used to display values for the user and *put-var* objects which are used to write to global variables. Since they are very similar the put-var classes are subclasses of the output classes. Figure 5.5 shows the 0-out-1-in class hierarchy.

5. Implementation using G2

This chapter describes the parts of the implementation that are G2 specific. The chapter is quite short since we have focused our studies on general implementation issues.

G2 automatically makes a graphical representation of all instances. This has saved a lot of programming.

The user interface in the implementation has a main workspace (a window) called *user workspace*. This is where tasks and programs (hierarchical function blocks) are placed. There are also buttons for toggling between local and global sort. Each task has a subworkspace where the user controls the task. The hierarchical function blocks have subworkspaces where the programs are defined. Figure 5.1 shows the user interface. Class definitions and methods are placed on workspaces hidden for the user.

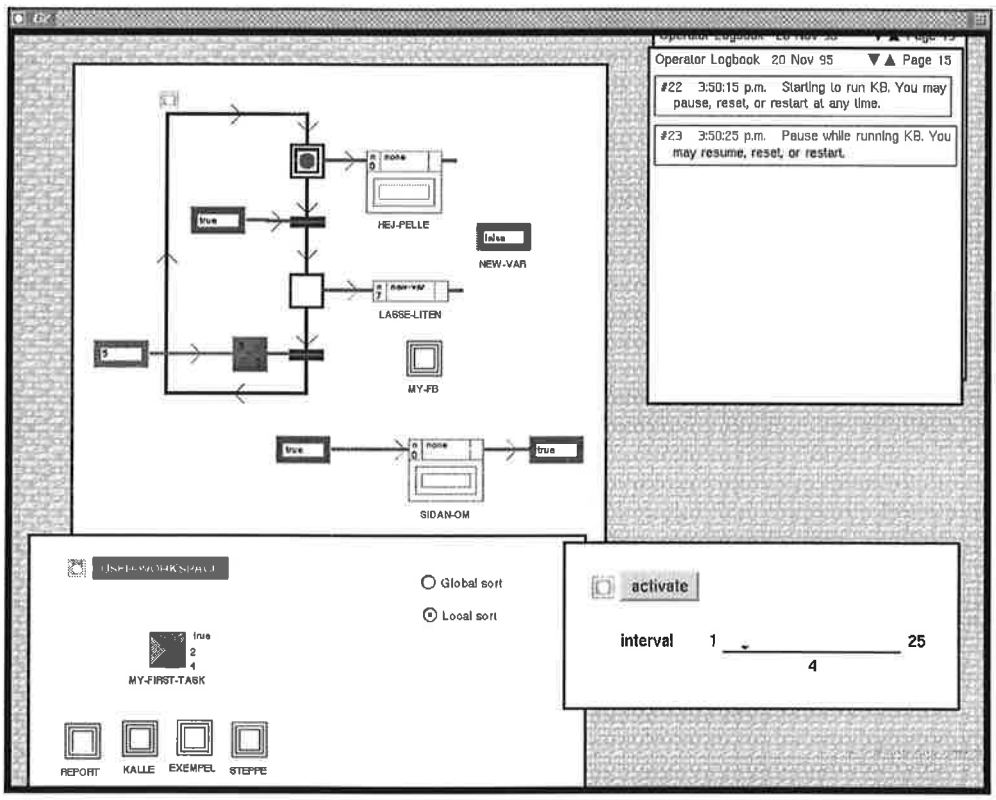


Figure 5.1 The user interface.

The implementation has the classes *My-object* and *My-connection* as the top classes. There are four main classes: *Task*, *Global-variables*, *Function-block* and *SFC-element*. They are described in the following sections. The class *User-function-block* works only as a mix-in class that makes a function block hierarchical. A user function block has a subworkspace where the hierarchical structure is defined. Figure 5.2 shows the main class hierarchy.

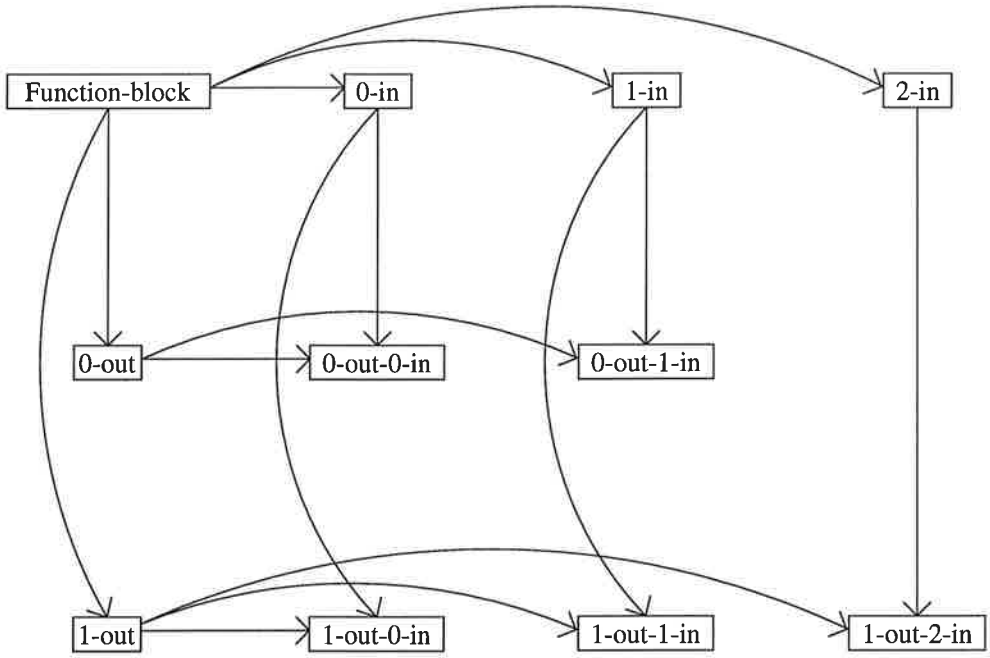


Figure 5.3 The function block matrix

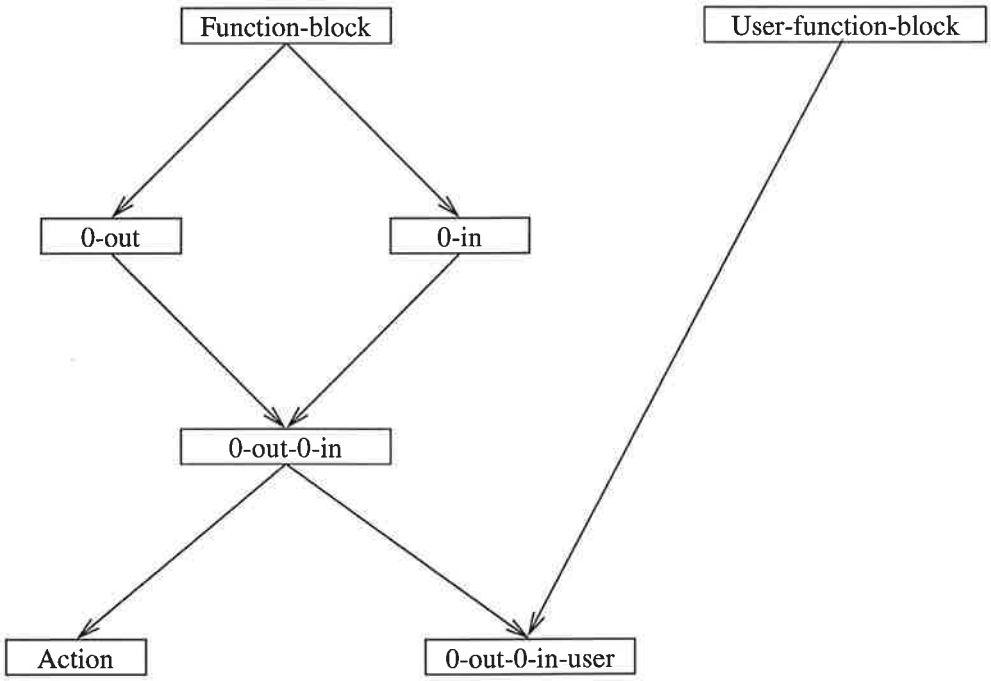


Figure 5.4 The 0-out-0-in class hierarchy

1-out-0-in

Classes with output but no input include *input* which are used to ask the user for a value and *get-var* which are used to read a value from a global variable. The class structure is analog with the one for 0-out-1-in and is shown in Figure 5.6.

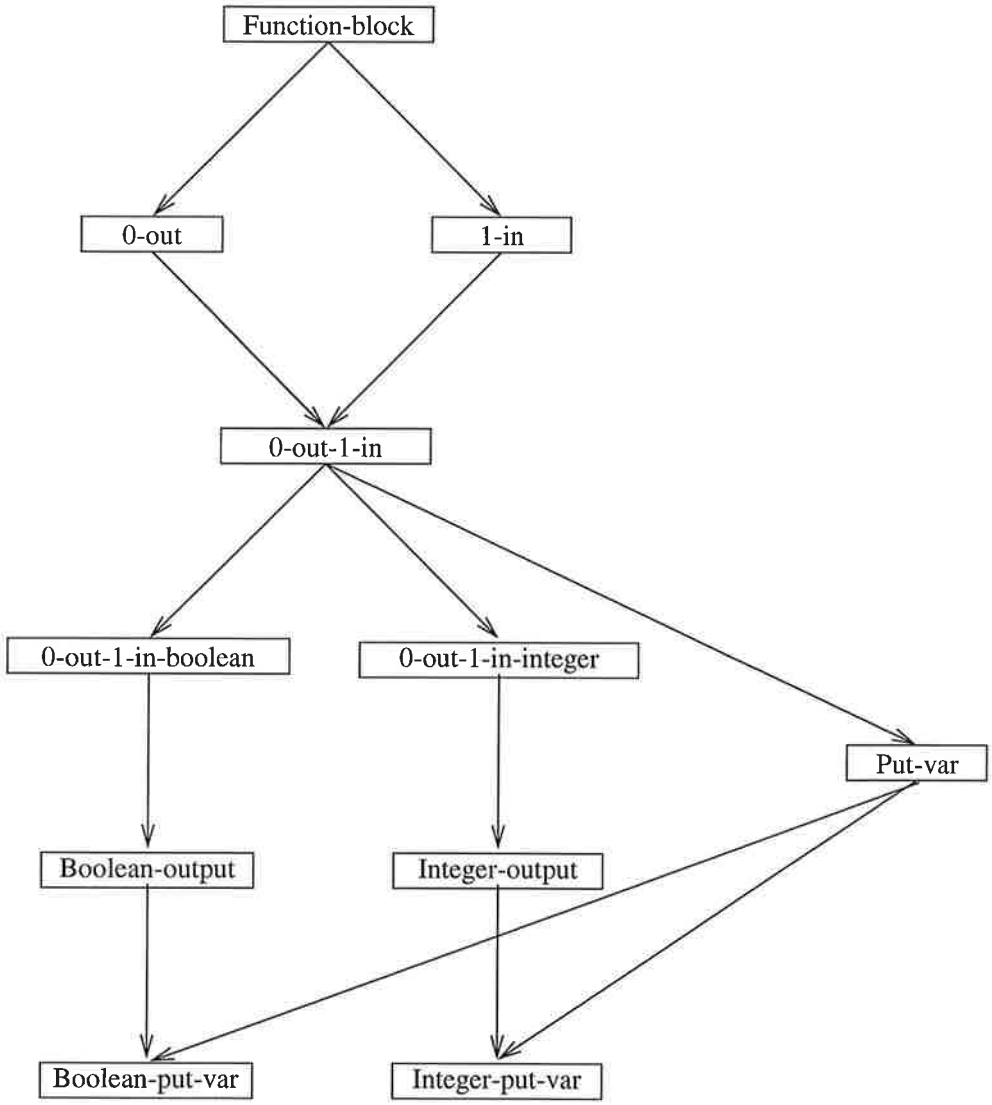


Figure 5.5 The 0-out-1-in class hierarchy

1-out-1-in

An inverter function has been implemented as an example of a boolean function and a counter as an example of a boolean to integer function. The loop function blocks inherit from *loop-function-block* since they need some special features and also from the 1-out-1-in classes since they can be seen as special function blocks. Since action blocks have a boolean input and output they are implemented as function blocks. They can also function as ordinary function blocks according to the standard. There are *action blocks* that are associated with a boolean actions and *user-action-blocks* where the user defines the action on the subworkspace of the action block. Figure 5.7 shows the 1-out-1-in class hierarchy.

1-out-2-in

We have implemented the function *and* as an example of a boolean function and the function *add* as an example of an integer function. We have also implemented a *1-out-2-in-integer-user* that is a hierarchical function where

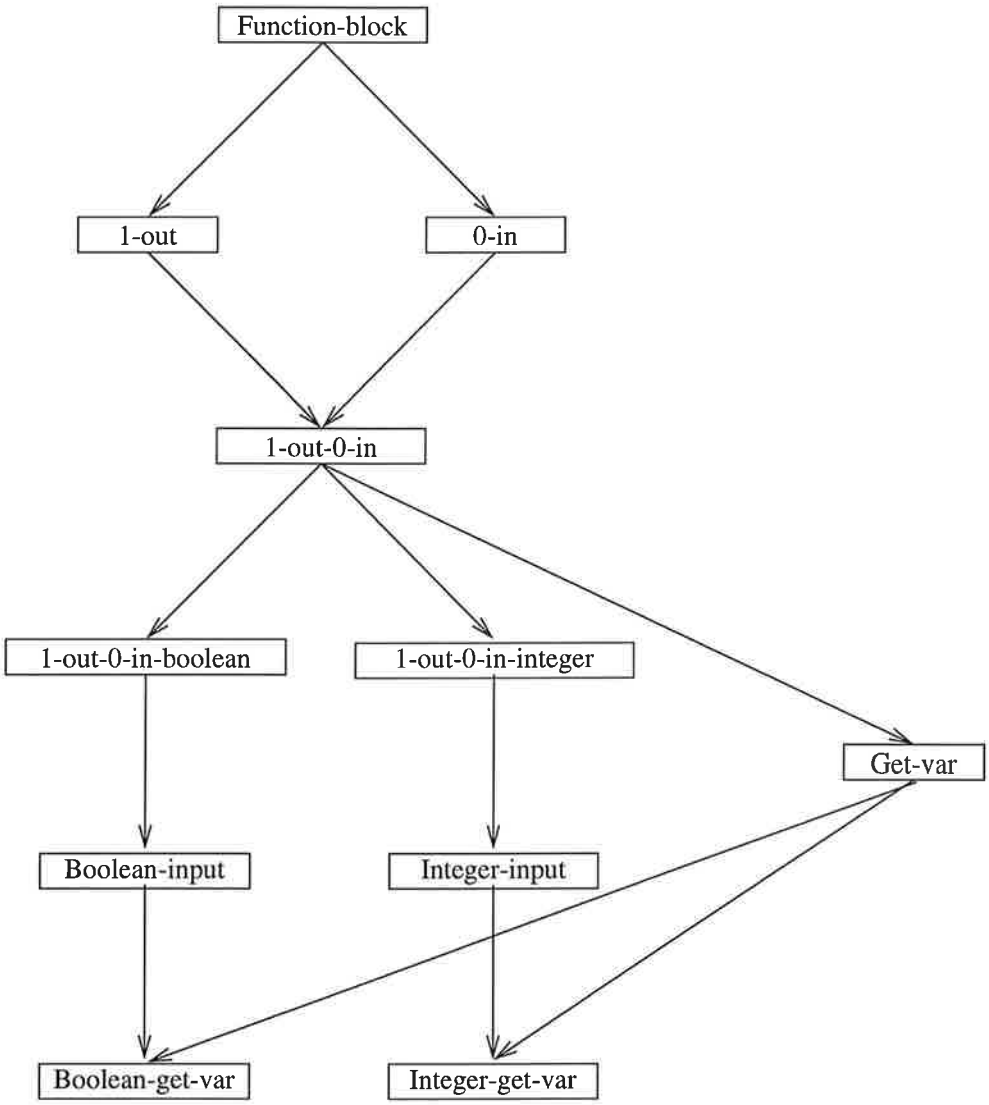


Figure 5.6 The 1-out-0-in class hierarchy

the user can define his own structure. Figure 5.8 shows the 1-out-2-in class hierarchy.

5.2. Hierarchical function blocks

In G2 it is not possible to define an internal structure of a function block in the class definition. This problem is solved by giving the user the possibility to define hierarchical function block to be a *type instance*. A type instance has two special methods: *make instance* and *update instances*

When using *make instance*, an instance is made by copying the type instance. The user gives the new instance a name. Then the function blocks of its subworkspace is given a name built together from the name of the new instance and the name of the corresponding function block on the subworkspace of the type instance. Figure 5.9 shows the type instance of the hierarchical function block FB2 in Figure 4.2.

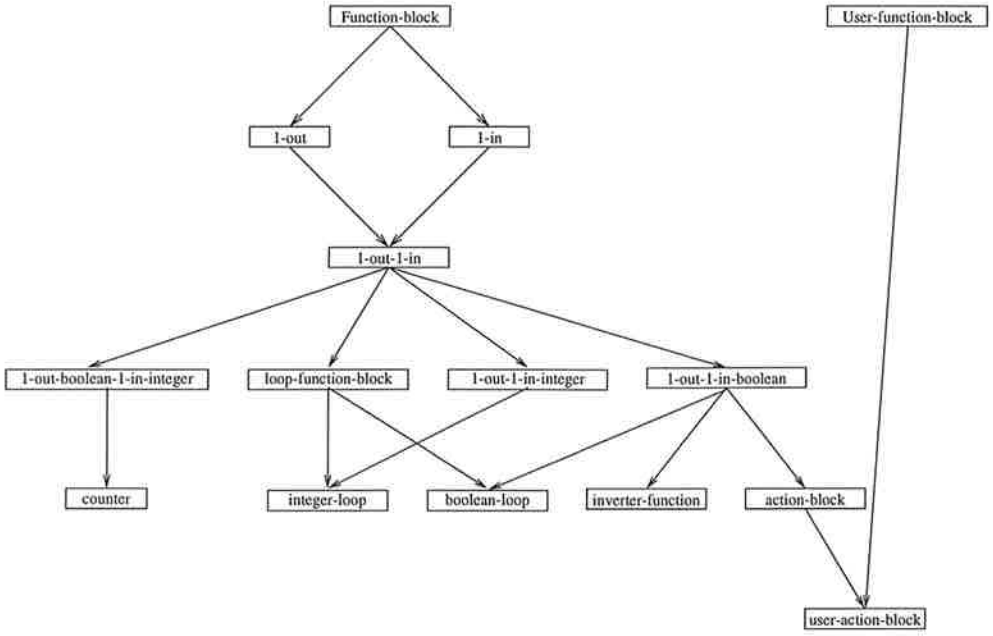


Figure 5.7 The 1-out-1-in class hierarchy

Each type instance has an attribute that is a list of all its instances. The method *update instances* updates the instances by replacing their subworkspaces with copies of the subworkspace of the type instance.

5.3. Sequential Function Charts

The SFC elements are steps, transitions and branches. Branches are used to build alternative and parallel paths. The SFC class hierarchy is shown in Figure 5.10.

5.4. Connections

To connect objects graphically we have used G2's *connections*. The advantage is that it is very easy to refer to associated objects with statements like "The step connected to the out-port of the transition ...". We have made different types of connections in different colors. This means that G2 does not allow the user to connect things in an illegal way. We have defined two types of connections: connections for dataflow and connections within an SFC. They are described in the following subsections. The connection class hierarchy is shown in Figure 5.11.

Data flow connections

There is one type of data flow connection for each data type, e.g. integer and boolean. Each data type has a unique color. A function in G2 that has stubs for connection must have the color of the connection specified. Overloaded functions (functions working on more than one data type) are therefore not implemented.

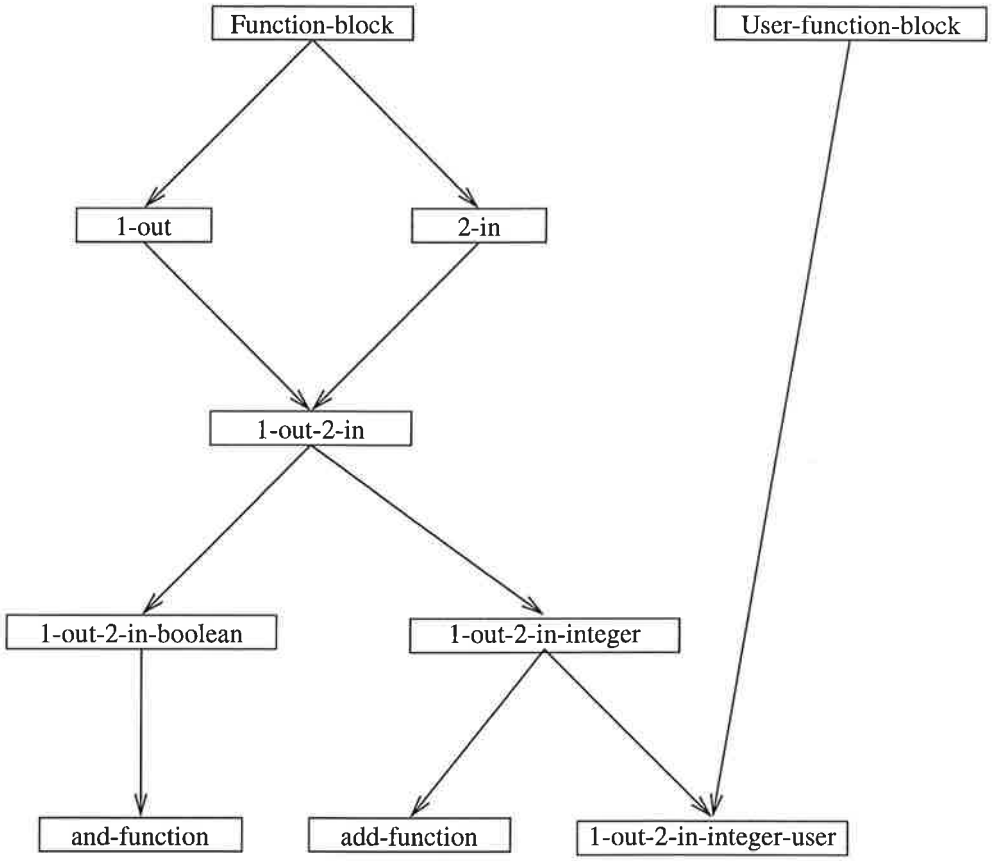


Figure 5.8 The 1-out-2-in class hierarchy

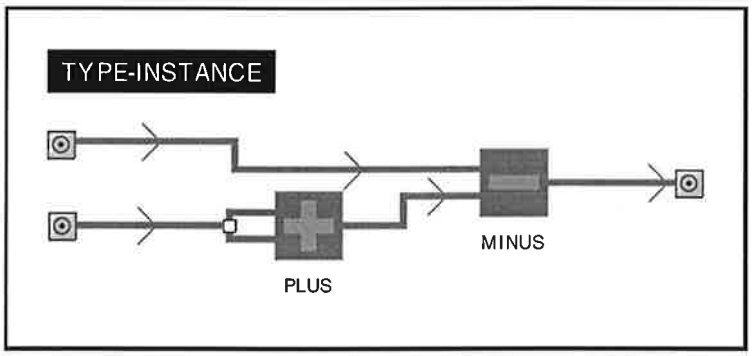


Figure 5.9 A type instance

SFC connections

Two types of SFC connections are implemented. They are step-transition-connections and transition-step-connections. They are used to connect steps, transitions and branch elements. The advantage with having two different connections is that the user can not build alternative and parallel paths in a wrong way.

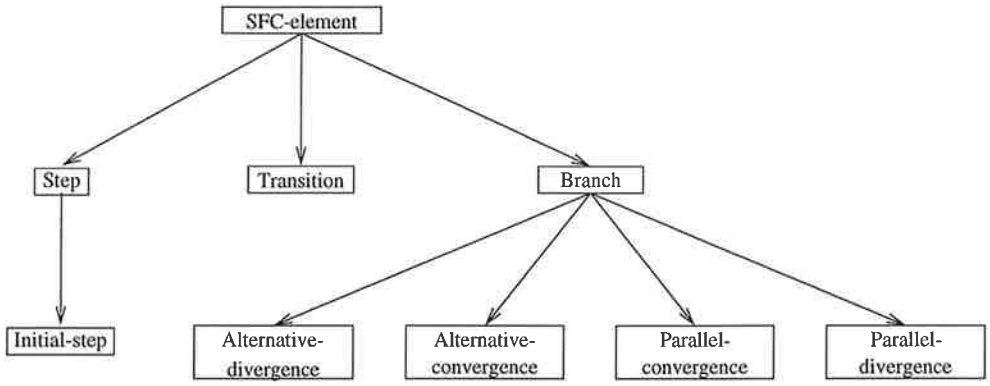


Figure 5.10 The SFC class hierarchy

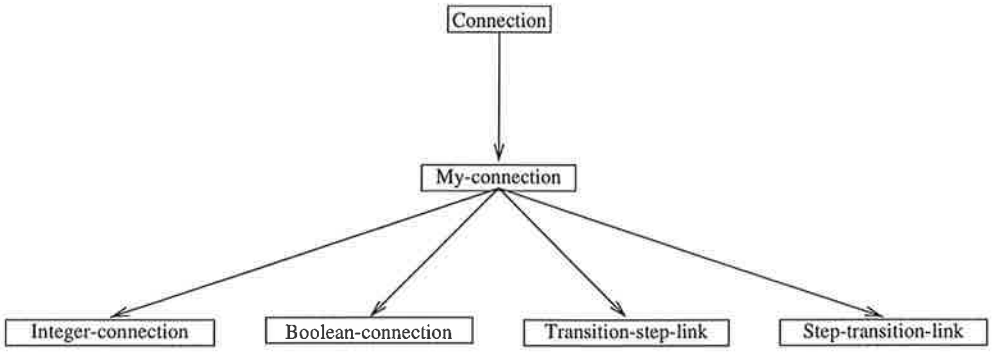


Figure 5.11 The connection class hierarchy

5.5. Global variables

A global variable has a specific data type, e.g. integer or boolean. Since an action is a boolean variable it is implemented as a subclass to boolean global variables. To get information of the state of an action the user uses a normal get-var function. Since the action shall be inserted in the execution list and executed it is also a function block. Figure 5.2 shows this relation.

5.6. Execution

The execution of a program follows the algorithm described in Section 4.3. When a task is initiated lists are build up for each hierarchical function block that is controlled by the task. Each function block is sorted into a function block list. Each step is inserted in a step list at an arbitrary order. When a task executes a hierarchical function block it calls its execution method. This method first checks which steps in the step list to update and then updates them. Then then each function block in the function block list is executed. Figure 5.12 shows the execution method.

5.7. Tasks

We use only one class to describe a task. The function blocks store their task as a *variable with an index*. G2 can then very effectively find all

```
exec-method ( myself: class user-function-block )

    x : class function-block;
    y : class connection;
    z : class step;

begin

    for z = each step in the step-execution-list of myself do
        call check-method (z);
    end;

    for z = each step in the step-execution-list of myself do
        call exec-method (z);
    end;

    for x = each function-block in the execution-list of myself do
        call input-method (x);
        call exec-method (x);
        call output-method (x);
        for y = each connection connected to x do
            call propagate ( y );
        end;
    end;

end
```

Figure 5.12 The execution method of a hierarchical function block

function blocks connected to a specific task and insert them in a list with the function *g2-indexed-attribute-item-list*. This way of implementing the association between a task and its function blocks is very G2 specific. A more natural way of implementing it would be to let the task have a list with pointers to its function blocks. G2 does not have pointers, however.

It was easy to implement a task handler as a periodic rule. The task handler controls all the tasks. An explicit link was not necessary since G2 has the possibility to use expressions like "for every task do ... ".

G2 does not support preemptive scheduling. This means that when one task has started to execute, it will never be interrupted by another one.

6. An example

This chapter describes how FBD and SFC could be used to control a larger and slightly more realistic example. The process is simulated in G2. Figure 6.1 shows the process that contains a tank, a pump, a valve and a heater.

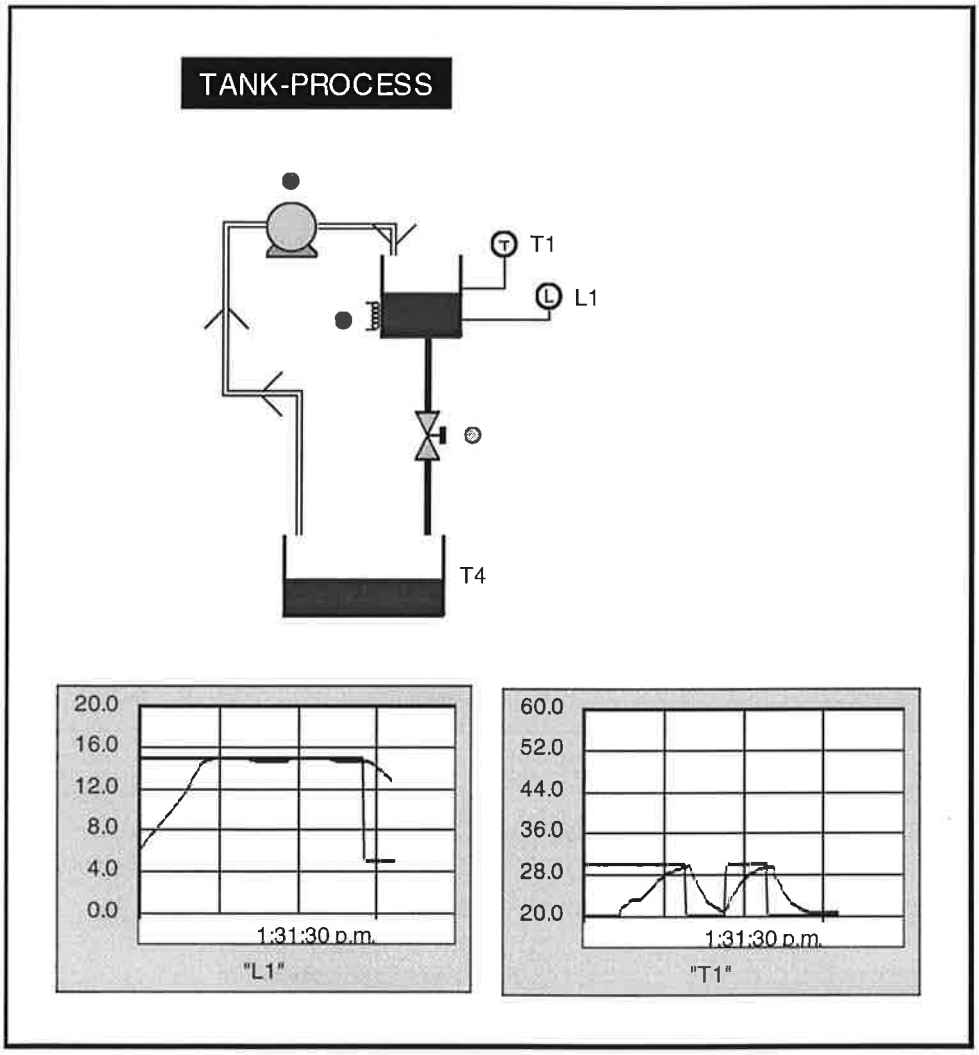


Figure 6.1 The tank process

The example contains two PID-controllers, one to control the level of the tank and one to control 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, as shown in Figure 6.2.

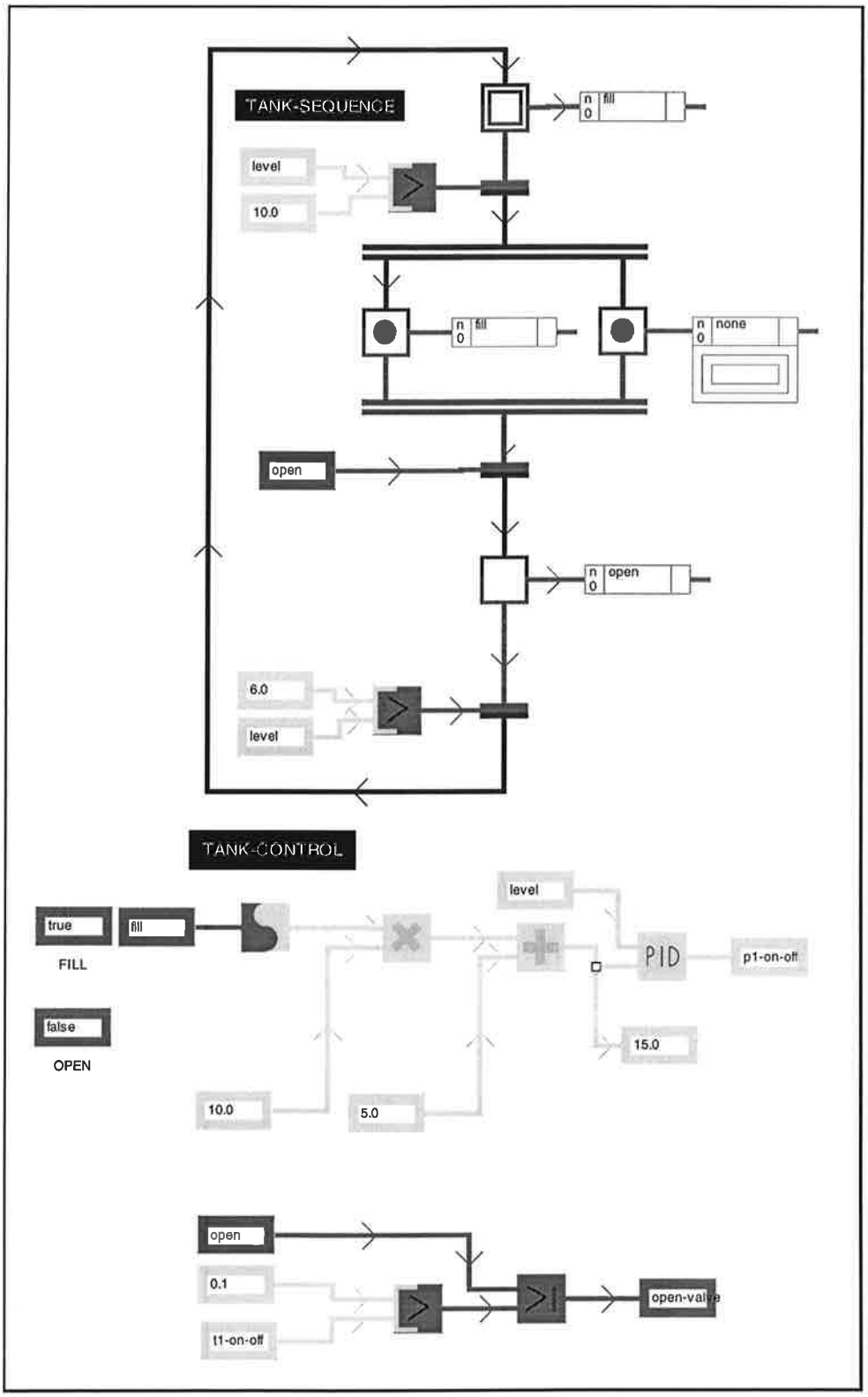


Figure 6.2 The main process control

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 inactivated 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 step to the right has an action defined on the subworkspace of the action block. This subworkspace is shown in Figure 6.3

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 a reference signal with the square wave shape in the right diagram of Figure 6.1. 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.

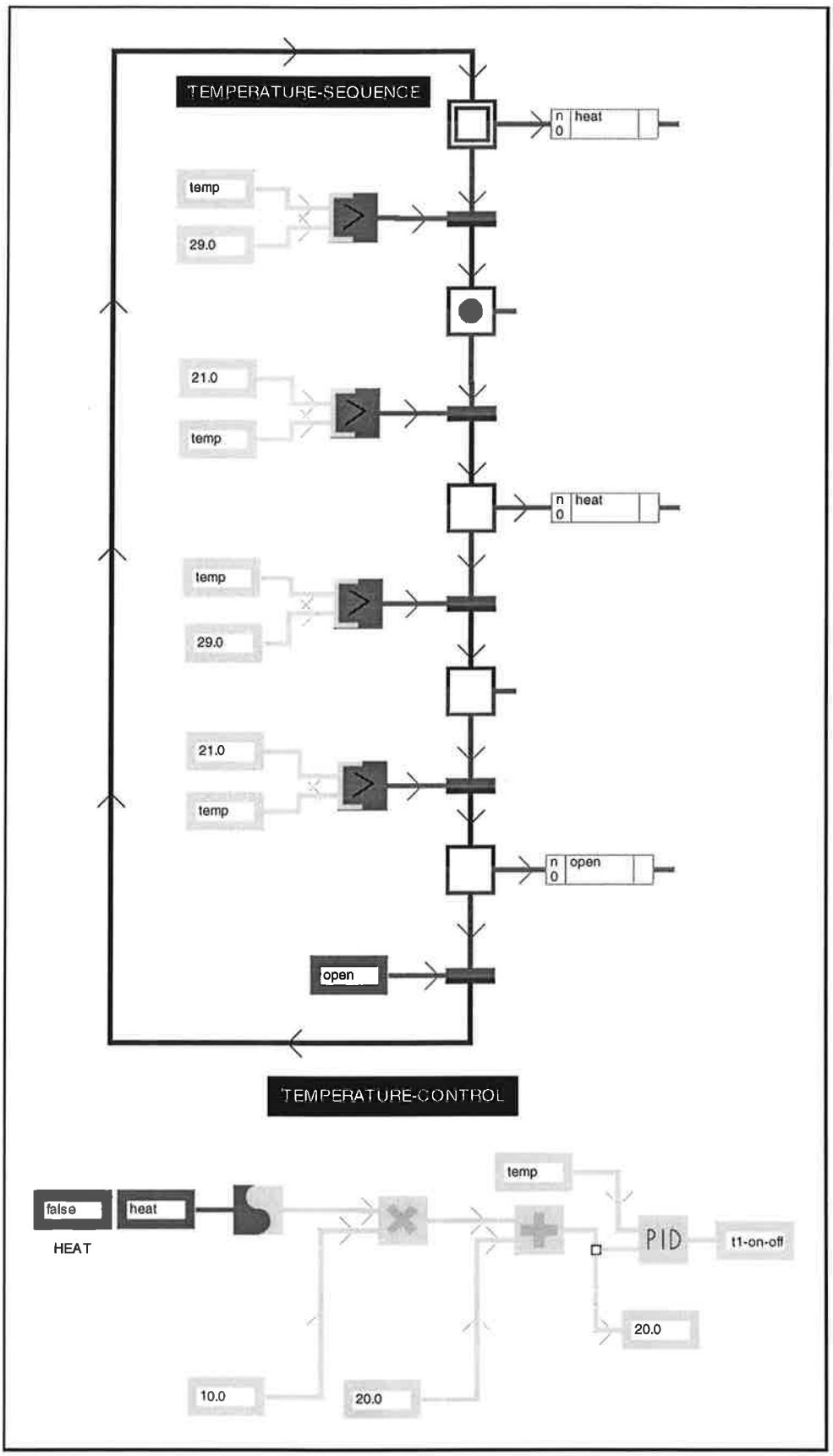


Figure 6.3 The control of the temperature sequence

7. Conclusions

The standard IEC 1131-3 contains most of the needed information 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 the standard. We believe that parts of a FBD connected to transitions should be executed before an SFC and parts associated with actions after an SFC.

All function blocks are sorted into lists before they are executed. Algorithms for sorting the lists locally or globally are implemented. Local sort was found to be easier to implement.

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 much faster with this method. It would be interesting to implement methods for generating such procedures.

The programming environment G2 is very good to use 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 to control real PLCs.

8. Bibliography

[1] IEC 1131-3, first edition, mars 1993.

[2] IEC 1131-3, Guidelines for the application and implementation of programming languages for programmable controllers, May 1995.

[3] Gensym Corporation, G2 Reference Manual Version 3.0, July 1992.

[4] Gensym Corporation, G2 Version 4.0 Beta Release Notes, March 1995

[5] Gensym Corporation, G2 Reference Manual Version 4.0, September 1995.

[6] Computer Implementation of Control Systems, Lars Nielsen and Karl-Erik Årzén, Department of Automatic Control, Lund, August 1995.

A. G2

This chapter is taken from [6]. Minor changes have been made to describe the new features in version 4.0 of G2.

G2 (trademark of Gensym Corporation) is the most widely spread and technically most advanced real-time expert system tool available today. It is written in Common Lisp which is automatically translated into C. It runs on a variety of UNIX and Windows platforms.

The main parts of G2 are: the knowledge-base, a real-time inference engine, a procedure language, a simulator, the development environment, the operator interface, and optional interfaces to external on-line data servers. The normal way of using G2 is to create a knowledge-base for a desired application off-line, and then run this knowledge-base in real-time.

A.1. Classes and objects

G2 is a strictly object oriented programming environment. This means that all components in G2, including rules, procedures, graphs, buttons, objects etc., are items. The items are organized into a hierarchy. All items have a graphical representation through which they are manipulated by mouse and menu operations. Operations exist for moving an item, cloning it, changing its size and color etc. The only type of item that the user has full control over, are the G2 objects. The user may define and manipulate objects in order to create the data structure for a certain application.

Objects are used to represent the different concepts of an application. They can represent arbitrary concepts, i.e. both physical concepts such as process components, and abstract ones. The objects are organized into a lattice of classes. Multiple inheritance is allowed. The class definition, or using G2 terminology, the object definition, defines the attributes that are specific to the class and the look of the icon. Icons can be created with an interactive icon editor. The attributes describe the properties of the object. The values of the attributes may be

- constants,
- variables,
- parameters,
- lists, vectors, or
- other objects.

Constants can be numbers, symbolic values, logical values (i.e. true or false), and text strings. Variables or parameters are used to represent entities whose values change during run-time. Variables are defined from four basic predefined classes:

- quantitative variables, i.e. integer- or real-valued variables,

- symbolical variables,
- logical variables, and
- text variables.

Parameters can be classified in the same way. The main difference between variables and parameters is that a parameter always has a value, whereas a variable must explicitly be assigned a value one way or another. A variable also has a validity interval, which specifies for how long a newly assigned value of the variable will be valid.

Lists and vectors may contain arbitrary values. The allowed values in a list or a vector can be specified. It is possible to have objects as the values of attributes in other objects. In that case, the attribute objects have no iconic representation.

Objects can be static, i.e., they are explicitly created by the developer, or dynamic, i.e., they are created dynamically during run-time. Dynamic objects can also be deleted during run-time. The G2 language contains actions to move, rotate, and change the color of an object. Using this, animations can be created.

Composite objects, i.e., objects that have an internal structure composed of other objects, can be created using objects as the value of attributes. It is, however, not possible to at the same time have a iconic representation for these objects. If such a representation is desired this has to be implemented using the subworkspace concept. In G2 each object and most items may have an associated subworkspace. On this (sub-)workspace arbitrary items may be positioned. The internal structure of an object can be represented on its subworkspace. It is, however, not possible to define that an object should have an internal structure of this type in the class definition. In this theses the problem is solved by defining certain instances as type instances. See Section 5.2.

A.2. Relating objects

G2 has different ways of defining relations between objects. One way is to let an object have attributes that are lists containing other objects. An object can be a member of any number of lists, thus very complex relations between objects can be defined this way.

A more direct way of relating two objects is to use connections. These are primarily used to represent physical connections, e.g., pipes or wires. It is, however, also possible to let connections represent abstract relations among objects. Connections have a graphical representation and may have attributes. They are defined in terms of a connection hierarchy. Both unidirectional and bidirectional connections are allowed. Connections can be used in G2 expressions for reasoning about interconnected objects in a variety of ways. A connection is attached to an object either at a pre-specified location, a port, or anywhere on the object. Connections can, like objects, be either static or dynamic.

A third way of relating objects is to use a special type of G2 item called relations. These can only be created at run-time and have no graphical

representation. They have no corresponding relation hierarchy and can not have attributes. Relations can be specified as being one-to-one, one-to-many, many-to-one, and many-to-many. They may actually relate any kind of items, not objects only. Relations can be used in G2 expressions in a similar way as connections.

A.3. The inference engine

G2 rules can be used to encapsulate an expert's heuristic knowledge of what to conclude from conditions and how to respond to them. Five different types of rules exist.

- **If** rules
- **When** rules
- **Initially** rules
- **Unconditionally** rules
- **Whenever** rules

When rules are a variant of ordinary **If** rules that may not be invoked through forward chaining or cause backward chaining. **Initially** rules are run when G2 is initialized. **Unconditionally** rules are equivalent to **If** rules with the rule conditions always being true. **Whenever** rules allow asynchronous rule firing as soon as a variable receives a new value, fails to receive a value within a specified time-out interval, when an object is moved, or when a relation is established or deleted.

The rule conditions contain references to objects and their attributes in a natural language style syntax. Objects can be referenced through connections with other objects. G2 supports generic rules that apply to all instances of a class. The G2 rule actions makes it possible to conclude new values for variables, send alert messages, hide and show workspaces, move, rotate, and change color of icons, create and delete objects, start procedures, explicitly invoke other rules, etc. G2 rules can be grouped together and associated with a specific object, a class of objects, or a user-defined category. This gives a flexible way of partitioning the rule-base. The following is an example of a G2 rule that applies to all instances of the class `water-tank`.

```
for any water-tank
 if the level of the water-tank < 5 feet and
 the level-sensor connected to the water-tank is working
 then conclude that the water-tank is empty
 and inform the operator that
 "[the name of the water-tank] is empty"
```

The real-time inference engine initiates activity based on the knowledge contained in the knowledge base, simulated values, and values received

from sensors or other external sources. In addition to the usual backward and forward chaining rule invocation, rules can be invoked explicitly in several ways. First, a rule can be scanned regularly. Second, by a focus statement all rules associated with a certain focal object or focal class can be invoked. Third, by an invoke statement all rules belonging to a user defined category, like safety or startup, can be invoked.

Internally the G2 inference engine is based on an agenda of actions that should be performed by the system. The agenda is divided into time slots. After execution, scanned rules are inserted into the agenda queue at the time slot of their next execution. Focus and invoke statements causes the invoked rules to be inserted in the agenda at the current time slot.

A.4. Procedures

G2 contains a Pascal-style programming language. Procedures are started by rule actions. Since G2 is a object oriented it also contains methods. These operate like normal procedures but are only defined for a specific class. Procedures are reentrant and each procedure invocation executes as a separate task. Procedures can have input parameters and return one or several values. Local variables are allowed within a procedure.

The allowed procedure statements include all the rule actions, assignment of values to local variables, branching statements (**If-then-else** and **case**), iteration statements (**repeat** and **for**), **exit if** statements to exit loops, **go to** statements, and **call** statements to call another procedure and await its result. The **for** loops may be either numeric or generic for a class, i.e., they execute a statement or set of statements once for each instance of the class.

Procedures are executed by G2's procedure interpreter. The procedure interpreter can not be interrupted by other G2 processing, i.e., the inference engine or the simulator. Other processing is only allowed when the procedure is in a wait state. A wait state is entered when a **wait** statement is executed, when the statement **allow other processing** is executed, and when G2 collects data from outside the procedure for assigning to a local variable.

A.5. Simulation

G2 has a built-in simulator which can provide simulated values for variables. The simulator is intended to be used both during development for testing the knowledge base, and in parallel during on-line operation. In the latter case, the simulator can be used, e.g., to implement filters for estimation of signals that are not measured.

The simulator allows for differential, difference, and algebraic equations. The equations can be specific to a certain variable of apply to all instances of a variable class. Each first-order differential equation is integrated individually with individual and user-defined step sizes. The numeric integration algorithms available are a simple forward Euler algorithm with

constant step size and a fourth order Runge-Kutta algorithm, also with fixed step size. GSPAN, an interface between G2's simulator and external simulators is available as a separate product.

A.6. Development interface

G2 has a graphics-based development environment with windows (workspaces), popup menus, and mouse interaction. Input of rules, procedures, and other textual information is performed through a structured grammar editor. The editor prompts all valid next input statements in a menu. Using this menu the majority of the text can be entered by mouse-clicking. It is, however, also possible to use the keyboard in an ordinary way. The editor has facilities for Macintosh style text selection, cut, paste, undo, redo, etc.

The Inspect facility allows the user to search through the knowledge base for some specified item. The user can go to the item, show all matching items on a temporary workspace, write them out on a report file, highlight them, and make global substitutions.

G2 has facilities for tracing, stepping, and adding breakpoints. The internal execution of G2 can be monitored using meters.

A.7. End-user Interface

G2 has facilities for building end-user interfaces. Colors and animation can be used. An object icon is defined as a set of layers whose colors can be changed independently during run-time. The meta-color transparent makes it possible to dynamically hide objects. Different user categories can be defined and the behavior with respect to which menu choices that are allowed can be set for each category. It is also possible to define new menu choices.

G2 contains a set of predefined displays such as readouts, graphs, meters, and dials that can be used to present dynamic data. G2 also has a set of predefined interaction objects that can be used for operator controls. Radio buttons and check boxes can be used to change the values of symbolical and logical variables by mouse clicking. An action button can be associated with an arbitrary rule action which is executed when the button is selected. Sliders can be used to change quantitative variables and type-in boxes are used to type in new variable values. Figure A.1 shows a G2 user interface with class definitions, instances, the class tree, a method, a button, a rule and a logbook page.

A.8. External interfaces

G2 can call external programs in four different ways: using foreign function calls, and using GFILE, GSPAN, and GSI. On some platforms, external C and Fortran functions may be called from within G2. GFILE is an

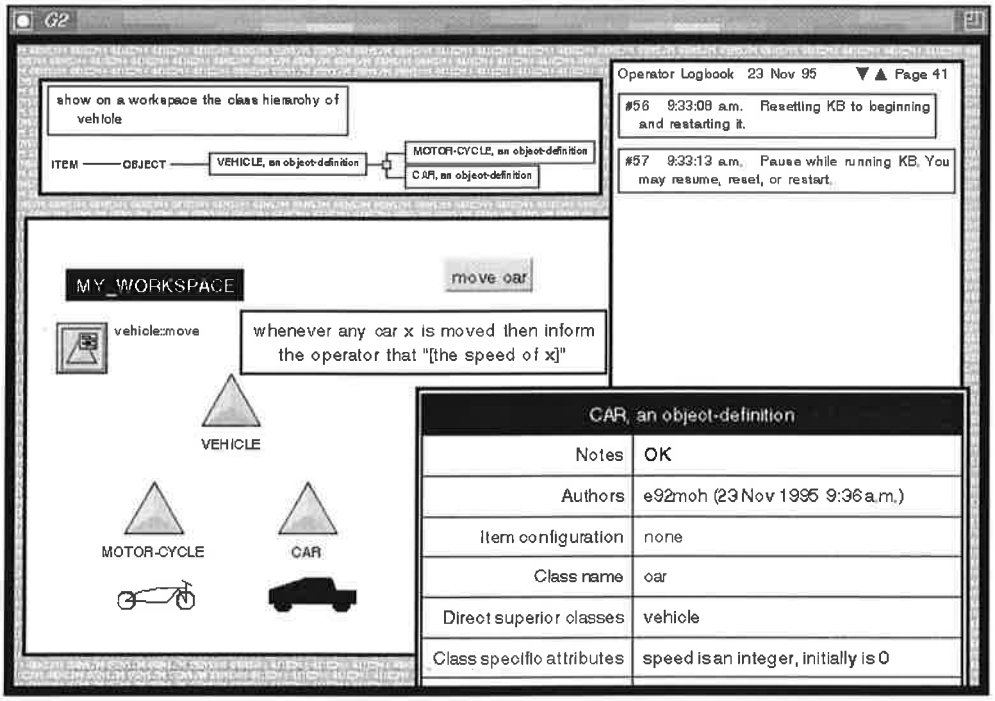


Figure A.1 A G2 environment

interface to external data files that allows G2 to read sensor data from the files. GSPAN is the interface between G2 and external simulators. GSI is Gensym's standard interface. It consists of two parts; one part written in Lisp that is connected to G2 and one part written in C to which the user can attach his own functions for data access. On the same machine, the two parts communicate using interprocess communication media such as pipes or mailboxes. On different machines, TCP/IP – Ethernet is used.

A.9. Drawbacks

The main problems with G2 stem from the fact that G2 is a closed system. G2 can only be interfaced with other program modules through the predefined interfaces. The G2 environment in itself is also a quite closed world. It is impossible to modify the way that G2 operates internally. If what G2 provides in terms of, e.g., graphics, class – object structures, etc., is insufficient, nothing can be done about it.

G2 can not be modularized. Hence, it requires quite powerful computers even if only a small subset of the functionality is used within an application.

Although G2 is fast compared to many expert system tools, it can be too slow for certain applications.For applications that require a fast response, G2 is inadequate.

B. Implementation files

To run the implementation you need G2 version 4 or later. A color monitor is highly recommended.

The program file is called IEC1131-3.kb. The modules sys-mod.kb and uilroot.kb are also needed.