ISSN 0280–5316
ISRN LUTFD2/TFRT--5470--SE

Object-oriented Structuring of Heating, Ventilation, and Air-Conditioning Systems

Magnus Strandh
Anders Ströbeck

Department of Automatic Control
Lund Institute of Technology
February 1993

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

Document name
Master Thesis

Date of issue
February 1993

Document Number
ISRN LUTFD2/TFRT--5470--SE

Author(s)
Magnus Strandh and Anders Ströbeck

Supervisor
Karl-Erik Årzén and Anders Wallenborg, TA

Sponsoring organisation

Title and subtitle
Object-oriented Structuring of Heating, Ventilation, and Air-Conditioning Systems

Abstract

This thesis investigates the possibilities of using object-oriented structuring of the control system for heating, ventilation, and air-conditioning systems. Different structuring methods are described and evaluated. The thesis also investigates the feasibility of using function blocks and Grafcet-style sequential function charts for programming the control system. A prototype system has been implemented in G2.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
61

Recipient's notes

Security classification

Contents

Acknowledgements

During this master thesis our supervisor Karl-Erik Årzén at the Department of Control at Lund Institute of Technology, LTH, has given us valuable help and suggested solutions to different problems. Without Mr. Årzén's excellent knowledge about G2 the implementation of the system would have required much more time. Anders Wallenborg at Tour & Andersson AB, TA, in Malmö has given us information about general HVAC systems and also provided the documentation of the specific system that we have implemented. For this we would like to express our gratitude.

At TA we would also like to thank a number of persons who have answered our questions about HVAC systems, among others, Per–Göran Persson, Mikael Krantz, and Lars Halling.

Leif Andersson at the Department of Control has given us valuable help with TEX, and questions concerning the computer system. Thank you!

Lund, February 1993

Magnus Strandh Anders Ströbeck

1. Introduction

The main purpose of this master thesis is to examine if it is possible to make an object-oriented structuring of a computer-based BMS[1]. Hopefully a well organized object structure will result in a faster, easier and therefore less expensive design procedure than the one used today. A major benefit is the possibility of reusing the control program or at least the main part of it, since most HVAC[2] systems have some basic features and thus perform in much the same way. The result of this object-oriented structuring will ideally be a graphically based design tool, that supports the design as well as the supervision of an HVAC installation. As an example the thesis includes an object-oriented implementation of a specific HVAC installation located in the office building of Tour & Andersson AB, (TA) in Malmö. TA manufactures computer-based BMSs for supervision, monitoring and control of HVAC systems. TA also provided us with information about general HVAC installations. When implementing the system we have used G2 (trademark of Gensym Corporation)[1] that supports object-oriented programming.

The secondary purpose of the thesis is to investigate the feasibility of using function blocks and Grafcet-style sequential function charts (SFC) for programming building management systems. Todays TA systems are programmed in a proprietary sequential programming language named IPCL. This programming style is in the thesis referred to as line programming. IPCL resembles to a large extent a conventional PLC language. There is currently a strong standardization trend among the automation and control system suppliers. Function blocks and Grafcet will clearly be a part of the emerging standard.

In Chapter 2 we discuss some advantages and disadvantages of conventional line programming and state-of-the-art object-oriented programming. In Chapter 3 we give a description of the existing TA SYSTEM 7. Chapter 4 describes the ideal system design environment and in Chapter 5 we present different alternative ways of object-oriented structuring of an HVAC system. Chapter 6 deals with function blocks and Grafcet. Since we cannot present a list of the program we have written using G2 we instead try to describe it in Chapter 7. We decided to describe the translation of the IPCL application program into Grafcet sequences in a separate chapter, and you will find a description of the different sequences in Chapter 8. Presenting data to an operator is not as straight forward as one might think. You have to consider carefully what a supervisor should or should not see or be able to change etc. We present some possible operator interfaces in Chapter 9. You will find our conclusions in Chapter 10 and in Appendix A there is a brief description of the real-time expert system G2, that we used when implementing "our" system. Appendix B describes how the function blocks were implemented using G2 and Appendix C contains the simulation equations that were used when simulating the implemented system in G2.

---

1 Building Management System

2 Heating, Ventilation, and Air-Conditioning

2. Background

In the HVAC world of today there exists two main approaches to the programming of HVAC control devices: *line programming* and *function-block programming* [2]. Line programming means writing control sequences in standard sequential line programming formats. Some languages used in line programming of DDC[3] systems look very much like high-level general computing languages, such as Pascal, and others look more like traditional PLC[4] languages, but certain additional functions specific for handling HVAC systems are added. Function-block programming makes it possible to reuse often occurring sections of a line program. The line program is divided into small program blocks where each block is handling a specific task. The blocks can be linked together and have parameters assigned by the programmer. A program is then written by assembling these preprogrammed blocks in various combinations.

One disadvantage of *line programming* is that end users, e.g. the caretakers of buildings, need specific training in the actual programming language [2]. The disadvantage, however, that speaks mostly against the method seems to be that software development for typical projects can be time consuming, by requiring entire programs to be rewritten for multiple systems, even though they all may operate very much the same. With a little help from copying features and editing aids the work can be reduced. In our opinion, this is not a very good solution since we think it is easy to make mistakes while developing a control sequence by copying parts from other programs. We also think that the copying and cutting procedure is not very smooth.

The primary advantage of *function-block programming* is its simplicity in standard applications [2]. A drawback of this approach is that whenever sequences are required that do not match available blocks, the programmer must employ custom blocks. This reduces the flexibility and makes the method satisfactory only when applied to standard HVAC applications. Therefore some people still prefer DDC systems employing line programming because they think they offer greater power and flexibility.

Today almost all modern building management systems are heavily software driven. In the last few years users and consulting engineers have become increasingly dissatisfied with the conventional line programming, and are therefore requesting much simpler systems [3]. The demand for a faster and easier way of designing new systems and programming the controllers has focused the attention on the state-of-the-art *object-oriented programming*. This approach has the potential to offer numerous advantages over traditional HVAC programming methods including: greatly improved ease of use and understanding, increased system reliability, greater flexibility and an increased ability to reuse parts of the control program. Object-oriented programming is also intuitive and simple enough that end users with no specific training in programming can actually program and configurate complex HVAC sequences [3].

---

3 Direct Digital Control

4 Programmable Logic Controller

3. The TA system

3.1 HVAC systems - an introduction

Air-handling including heat recovery is today requested in almost all new buildings – not only for the economical benefits but also for the improved indoor climate it provides. By also filtering the air and keeping the humidity at a pleasant level a clean and comfortable indoor climate can be achieved. These features are included in the concept HVAC systems. In the following text we will give a brief description of how a common type of HVAC system, seen in Figure 3.1, works.

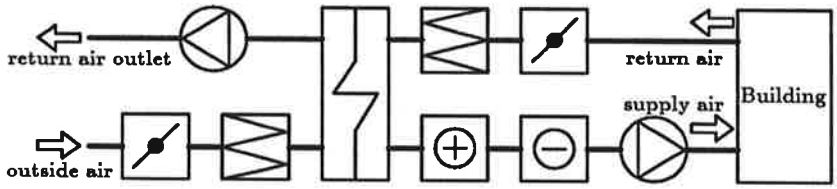


Figure 3.1 This is a common air-handling unit containing a heat exchanger, a heating coil, a cooling coil, a supply air fan, a return air fan, two dampers and two filters. An explanation of the symbols is found in Figure 3.2. The fresh supply air is distributed within the building and then the energy containing return air is returned. This energy is then the subject of recovery.

Outside air is fed into the AHU[5] where it is conditioned, either heated or cooled, depending on the outside temperature. The main task of the HVAC system is to keep the temperature as well as the pressure of the supply air and the return air at a constant level. Instead of controlling the temperature of the supply air it is also possible to control the temperature of the return air or the temperature in a room. In case of many rooms the average temperature is used as the feedback input to the controller. The pressure of the return air is also kept at a constant level. Heat or cold, depending on the status of the system, is recovered from the return air. In addition to the basic components, e.g., heat exchangers, heating coils, cooling coils, fans, dampers and filters, an HVAC system also contains control devices such as temperature and pressure sensors, PID controllers, etc.

3.2 A description of basic air handling equipment

Here follows a general description of some basic components that can be part of an AHU[4]. The graphical symbols representing the components, according to the Swedish standard SS 03 22 60, are shown in Figure 3.2.

5 Air-Handling Unit

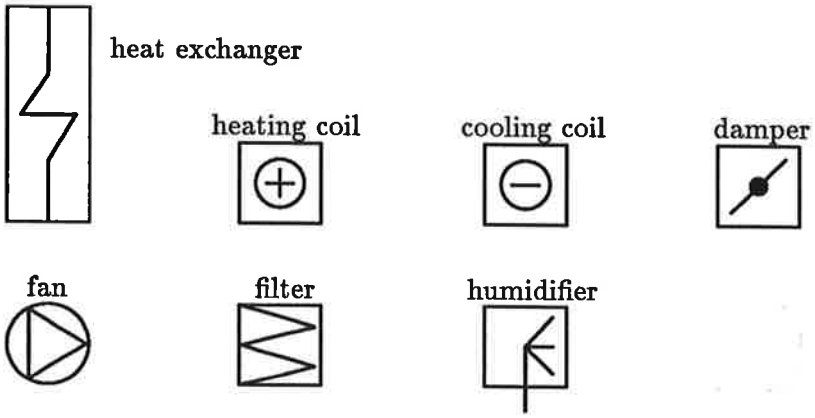


Figure 3.2 Symbols, for some basic air handling equipment, following the Swedish standard SS 03 22 60.

Air heat exchangers

Since the return air contains a lot of internal thermal energy, there is a great profit in using heat recovery. Flowing through a heat exchanger, the return air emits its energy passing by a heat-absorbing surface. Simultaneously the cold outside air is warmed up when passing by the same hot surface. The most common heat exchanger in HVAC applications is the *rotary regenerative heat exchanger*. The rotor is constructed of corrugated aluminum foil which has a large heat-absorbing surface. Thus the rotor serves as a heat transfer surface in direct contact with the air and through which heat is transferred by conduction. If the return air contains strong odours, e.g. smoke or solvent, it is better to use a *plate heat exchanger* which has separate air channels for the cold outside air and the return air. A more expensive variant of a plate heat exchanger is the *counterflow heat exchanger* whose advantage is its improved efficiency. There are several other types of heat exchangers such as *liquid-coupled heat exchanger*, *heat-pipe heat exchanger*, etc. It is also possible to recover heat by just mixing the cold air with the return air using a mixing damper but today this method, called *recirculation of air*, is a subject of discussion due to the discovery of "sick houses", allergies, etc.

Heating coils

If the heat exchanger does not manage on its own to warm up the cold outside air a heating coil is used. In a heating coil the air passes a hot surface in the same way as in a heat exchanger. The surface is heated either by hot water or by electricity thereby naming the two different main types of coils, the *hot water heating coil* and the *electrical heating coil.*

Cooling coils

The use of a cooling coil is necessary when the temperature of the outside air is higher than the required temperature indoors. Two main types of cooling coils are available, the *water cooling coil* and the *direct expansion coil.* The cooling medium in the two types is cold water and freon respectively. A cooling coil also serves as an air-dehumidifier when chilling the air to a temperature below its dew-point causing the water to condense.

Fans

Fans are used to produce a flow of air within the building and they can be divided into two common groups: *centrifugal fans* and *axial-flow fans*. In a centrifugal fan the impeller consists of fan blades positioned in a shell-shaped casing. An axial-flow fan, on the other hand, looks more like an aeroplane propeller and is mainly used in very large buildings.

Dampers

Sometimes you want to turn off the airflow to premises not in use, e.g. conference rooms. This is done with *dampers*. As mentioned earlier, dampers are used when recovering energy by recirculation of air, but the main usage is as cut-off dampers against the outside air. In case of fire, their task is to prevent smoke from dispersing within the building. Dampers are also often used to modulate the flow of air, e.g. in order to achieve correct distribution of air for different parts of the building.

Filters

Filters do not affect the climate condition of the air in the way that cooling coils, heating coils etc. do. Filters are used to purify the air, which today is very polluted in the cities. They also protect the different parts included in an installation.

Air-humidifiers

If the humidity needs to be higher than normal, due to the activity in the premises, the outside air has to be humidified. This can be achieved mainly in two different ways, by *steam dampening* or by *evaporation dampening*. An alternative to the latter method is called *spray dampening*, where water is atomized and sprayed into the air where it vaporizes. Advantages with dampening are, since dry air can irritate the respiratory tract, that inhalation of the air becomes more pleasant and the eyes of people wearing contact lenses will not get dry.

Air-dehumidifiers

If the humidity is too high, e.g. in store-rooms demanding a dry climate for the goods, the air has to be dehumidified. Dehumidifiers are also needed in climates with high temperature and humidity, where the humidity of the supply air gets too high when it is cooled. By using a cooling coil to cool the air below its dew-point the humidity will condense on the cooling surface. Another approach is used in the *sorption dehumidifier* where the air passes a rotor impregnated with a salt, absorbing the damp in the air.

3.3 Two examples of AHUs

There is a great variety of AHUs since they can be assembled by combining the basic components in many different ways. For instance, a system designed

for running in Kuala Lumpur does probably not include a heater! What components that are to be part of the plant is decided both by the climate where the system is running and, of course, by the demands of the customer. You can get very complex systems but also simpler ones whose main task is to recover heat from the return air and keep the indoor temperature at constant level. Below is shown two examples that illustrate the variety.

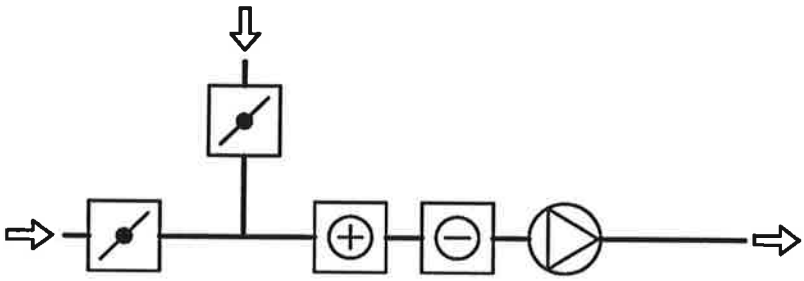


Figure 3.3 A simple air-handling unit only containing two dampers, a heating coil, a cooling coil, and a supply air fan. Heat recovery is obtained using recirculation of air.

Figure 3.3 shows a simple installation using recirculation of air to recover heat, while Figure 3.4 shows a more complex system using both a rotary regenerative heat exchanger and recirculation of air to recover heat. Furthermore there are filters for cleaning the air and a humidifier to dampen the air.

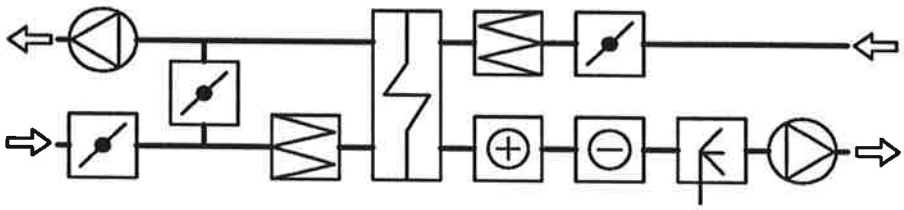


Figure 3.4 A complex air-handling unit containing: a rotary regenerative heat exchanger, a heating coil, a cooling coil, a humidifier, a supply air fan, a return air fan, dampers, and filters. Heat recovery is obtained using both recirculation of air and a heat exchanger.

3.4 A description of the examined HVAC system

In this thesis we have examined an HVAC system where the AHU contains three different components that control the temperature: a rotary regenerative heat exchanger, a hot water heating coil, and a direct expansion cooling coil, cf. Figure3.5. In most cases it is sufficient to run only the heat exchanger, but during winter it may also be necessary to use the heating coil. During summer, on the other hand, it might be so hot that the air has to be cooled with the cooling coil. If it becomes so hot outside that the temperature is lower inside the building the heat exchanger can be used to recover cold from the return air to cool off the incoming outside air, before the cooling coil does the rest of the job. Furthermore, there is a supply air fan with variable guide vanes controlling the pressure of the supply air and a return air fan, also with variable guide

vanes, controlling the pressure of the return air. The fans, of type centrifugal, are both operated by two electrical motors making it possible to run each fan at two different rpms, i.e. slow and fast. Apart from these items the system also includes some dampers and filters.

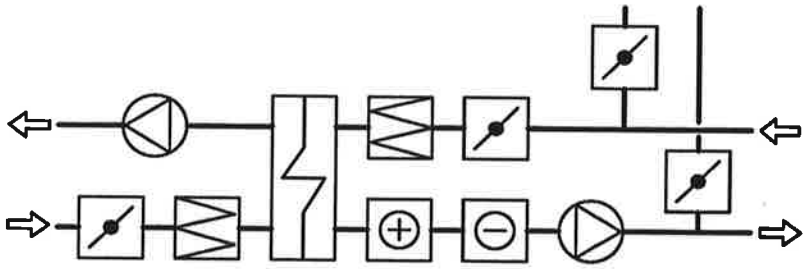


Figure 3.5 The examined air-handling unit. The only difference from the unit in Figure 3.1 is that this unit also contains two dampers used in case of fire.

The rooms are heated with conventional radiators. Each room in the building is also equipped with a so called VAV[6]-box, which controls the amount of supply air led into the room. The VAV-box contains a Zone-Controller (ZC) which keeps a constant temperature in the room by controlling a damper in the box, letting more or less air into the room. The climate of the room is therefore affected both by the radiator and the airflow from the VAV. Having a VAV at each room makes it possible to obtain an individual room temperature using a local thermostat, which provides the ZC with a setpoint.

3.5 Hardware – the TA SYSTEM 7

TA has specialized on designing a flexible computer-based building management control system for controlling and supervising HVAC installations of the type mentioned above. This has resulted in the TA SYSTEM 7, which is a networked, microprocessor-based DDC system for building automation.

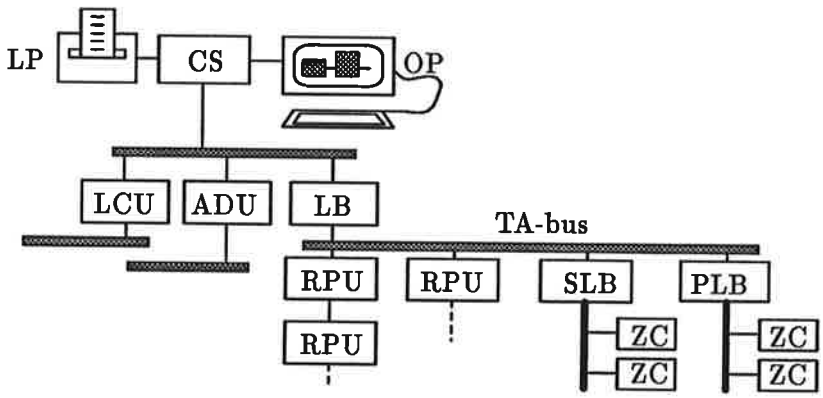


Figure 3.6 The organization of the hardware in the TA SYSTEM 7.

Figure 3.6 shows an overview of how SYSTEM 7 is organized. From the main computer, called Central System (CS), the operator (OP) can supervise a

6 Variable Air Volume

certain installation. The OP can, for example, check the regulator parameters in a specific regulator and also, if he wants, change them. He also has full information of the current status of the system, which devices are active, etc. An important task is to handle alarms due to failures in the system.

The Line Buffer (LB), a bus interface adapter between the CS and units connected to the TA-bus, makes it possible to connect different installations within the building to the same CS. The system also supports off-site access to installations in other buildings via telephone network connections. This is handled with a Line Concentrating Unit (LCU), an interface to permanent network connections, or an Automatic Dialing Unit (ADU) which serves as an interface to dial-up network connections. Via Pier Line Bridges (PLB) and Sub Line Bridges (SLB) communication between the zone-controllers and CS is possible.

The RPU[7] is a microprocessor based DDC stand-alone controller equipped with its own program and system functions for controlling, e.g., built in PID controllers, and analog/digital in- and outports. Each RPU runs independently of the others and is able to communicate not only with the CS but also with other RPUs within the system. The RPU controls the devices in a specific AHU sending signals to, e.g., start or stop the heating coil. In this thesis our attention is mainly focused on the programming of these controllers.

3.6 Software – programming the RPU

The RPU is programmed using conventional line programming. The language used when programming the control sequences is developed by TA and called IPCL[8]. The language supports traditional line programming for HVAC applications as described in Chapter 2, and the syntax is closer to a PLC language than Pascal. After the program is written it is compiled and downloaded into a RPU. In Figure 3.7 is shown a short sequence, from an IPCL program, controlling the return air fan.

The procedure when designing a new system is today carried out mainly in two steps. First an application expert receives the specification of the new system and decides what devices are to be part of it, based upon the demands from the customer. He also decides which operation modes are necessary and how the devices should be connected to perform the task. The next step is handled by a programmer and consists of developing the control program itself. This requires a rather skilled programmer with lots of experience in this kind of system.

The problem with reusing code is today solved in different ways. Some of the programmers use their favorite editor and a program library. The library contains smaller blocks which are put together in the editor to a complete control program. Another common way is to save entire projects in a program library. Then the programmer uses the program from a similar project as a base and only makes the necessary changes.

---

7 Remote Processing Unit

8 Interpretive Process Control Language

```
        ILV(5) SET ILV(12)
        IF IFV(15) >= SV(19) AND NOT ILV(5) AND NOT ILV(2) THEN
               IFV(22) + SV(7) = IFV(22)
        ELSE
               SV(6) = IFV(22)
        ENDIF
C
        IFV(22) >= SV(20) NJ TESTF1
C
COUNTF          R0(5)
                S1 ILV(22)
                SV(14) = IFV(15)
                IFV(9) + SV(7) = IFV(9)
                IFV(9) >= SV(9) NJ TESTF1
                S0 ILV(22)
                S1 ILV(5)
                SV(6) = IFV(9)
C
TESTF1  IFV(15) <= SV(18) NJ REGV
                S0 ILV(5)
```

Figure 3.7 An example, of an IPCL program, showing the syntax. In the figure the comments, that are always present in a program, are omitted due to lack of space. The comments improve the readability of the program sequence, which here is part of a sequence that control a return air fan.

4. The Vision

In this chapter our vision of an ideal future design environment for computer-based building management systems is presented. In our mind the design environment would be based on a graphical design tool. This tool should support the design of a BMS as well as debugging and supervision of the system. When using the program as a design tool the object corresponding to an actual, physical component – e.g. a heating coil – should be selected from a library, placed on the screen and connected to other objects. By representing the object with an icon that looks like the standard symbol of the physical component, the picture used when supervising the installation, i.e. the end-user interface, is drawn at the same time as the system is configured. When this is all done the functions that should be available in the system, e.g. alarm-handling or actions in case of fire, are defined by the designer. This is all that the designer should have to inform the program about before it automatically generates a control program for the air-handling unit in question and without any further efforts from the designer/programmer the system should now be ready to run.

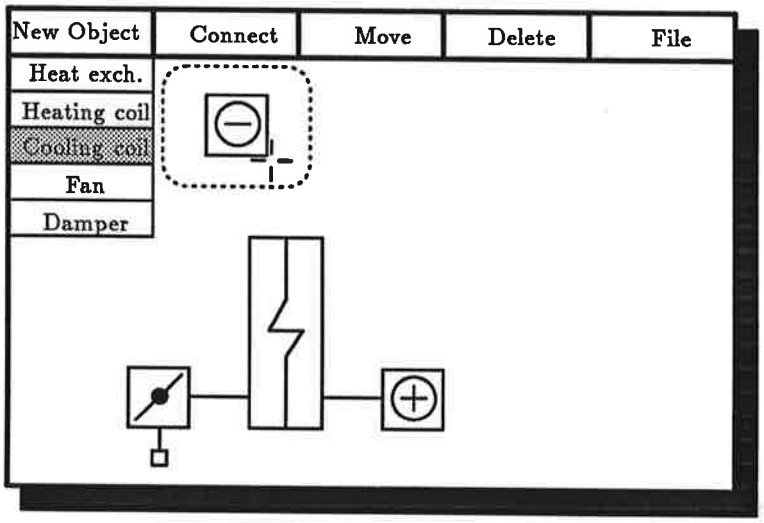


Figure 4.1 This is a picture of how the design of a BMS could look like when using a graphical design tool supporting some basic editing functions. The designer has placed a damper, heat exchanger and a heating coil on the screen and connected them to each other. The next object he has chosen is a cooling coil.

The course of action described above is of course just a vision of how one would like to design a BMS and as in most cases there are a number of problems to overcome in order to realize the vision. In this case the most important one is that the control functions of an object may depend on whether another object is present in the particular installation or not. Therefore a more reasonable way of creating a BMS is to have a library containing of a number of different control functions for each physical component. From this library the designer chooses the function that best agrees with his wishes and in some way relates this function to the objects on the screen. This of course calls for some more knowledge of how the installation works than it does in the vision above. Yet it seems to be a satisfactory compromise between the ideal world and reality.

5. Object-oriented structuring

Object-oriented programming can be viewed as a method of modeling a process with objects that correspond to actual items, events and actions within the process. In short, an object is code and data that behaves like something in the real world. If you can model data as an object containing both characteristics of the data item itself and operations allowed to be performed on the data, and then bring them together to a block, you have defined a class. Therefore a class can, to some extent, be described as a collection of abstract data types and operations that belong to an object. The class concept also includes the feature of inheritance, which makes the method powerful in some applications. You can find more interesting reading about object oriented structuring in the book *Object Oriented Design with Applications* [5].

5.1 Inheritance, classes and objects

Assume that you have a set of boxes that when you first look at them seem to have the same attributes, e.g. they all have a certain height, width and depth. You then create a class, "boxes", with these three common attributes, and every object (box) is an instance of this class. However when you study your objects more carefully you realize that there are some attributes that are applicable to some objects but not to others. For instance some of the boxes can be opened and the others not. To get a better grip of things, you then create two subclasses, "open" and "closed". They both inherit the attributes of the superior class, i.e. height, width and depth.

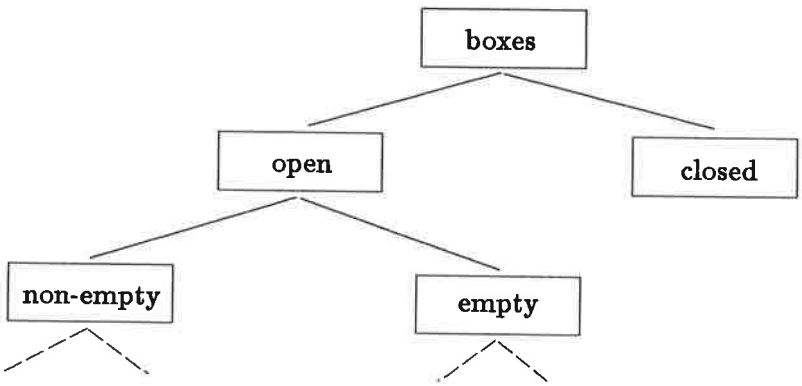


Figure 5.1 An example of a class hierarchy. The subclasses are a restriction or a specialization of the superior class. They also inherit the attributes from the superior class.

The difference, however, is that the boxes belonging to one of the subclasses can be opened and the ones that belong to the other can not. A further investigation of the boxes that can be opened shows that some of these boxes have

a content and the others do not. Based on this you can classify the boxes even better and create another two classes, "empty" and "non-empty", that are subclasses of "open". In this way you create a class hierarchy, a tree structure, cf. Figure 5.1. The example above shows how you reason when structuring a system in an object-oriented way. You define classes with certain attributes, that you arrange in a tree structure, where the subclasses inherit the attributes of the superior class. Thus the subclasses are in some way a restriction or specialization of the superior class. You may also redefine attributes inherited from the superior class if that should be convenient. To define the system you then create objects that are instances of the classes in the hierarchy.

5.2 Structuring the system

The purpose of the object-oriented structuring is to be able to develop a graphical interactive design tool for configuring the control program including control sequences, PID controllers etc. The first question that comes to your mind when you start thinking about the structuring is if you can do this in more than one way. The answer to this question is – of course – yes. One obvious way to do it is to divide the system into physical components, let us call this a *component-based structuring*. Another approach, which we have chosen to call *function-based structuring*, is to choose the classes from the control program's point of view. After studying some IPCL programs for different applications, written for the RPUs, we discovered that they all can be divided into more or less independent parts. The division of the control program is performed differently in the two structuring approaches. It is also possible to use a modified version of the component-based structuring together with the function-based structuring, and this approach will be referred to as *combined structuring*.

When discussing the three methods the terms *end-user interface* and *control interface* are mentioned. The end-user interface shows a graphical picture, containing the symbols of the physical components in the air handling plant plus dynamically updated displays showing, e.g., the supply air temperature, the supply air pressure, fan status, etc. The end-user interface is used when supervising the system. The control interface shows a picture of the control sequence, PID controllers, etc., and is used when debugging the system or as a part of the supervision of the system. The two different interfaces are further described in Chapter 9.

5.3 Component-based structuring

The first step in the component-based structuring is to identify the different physical components that are present in the system. For each component a corresponding class is defined and this results in a tree structure similar to the one in Figure 5.1. To be able to describe, e.g. a rotary regenerative heat exchanger, the class *Rotary-Heat-Exchanger* is defined, cf. Figure 5.2, and this class is probably a subclass of the class *Heat-Exchanger*. The advantage with this approach is that the classes in the system easily can be identified.

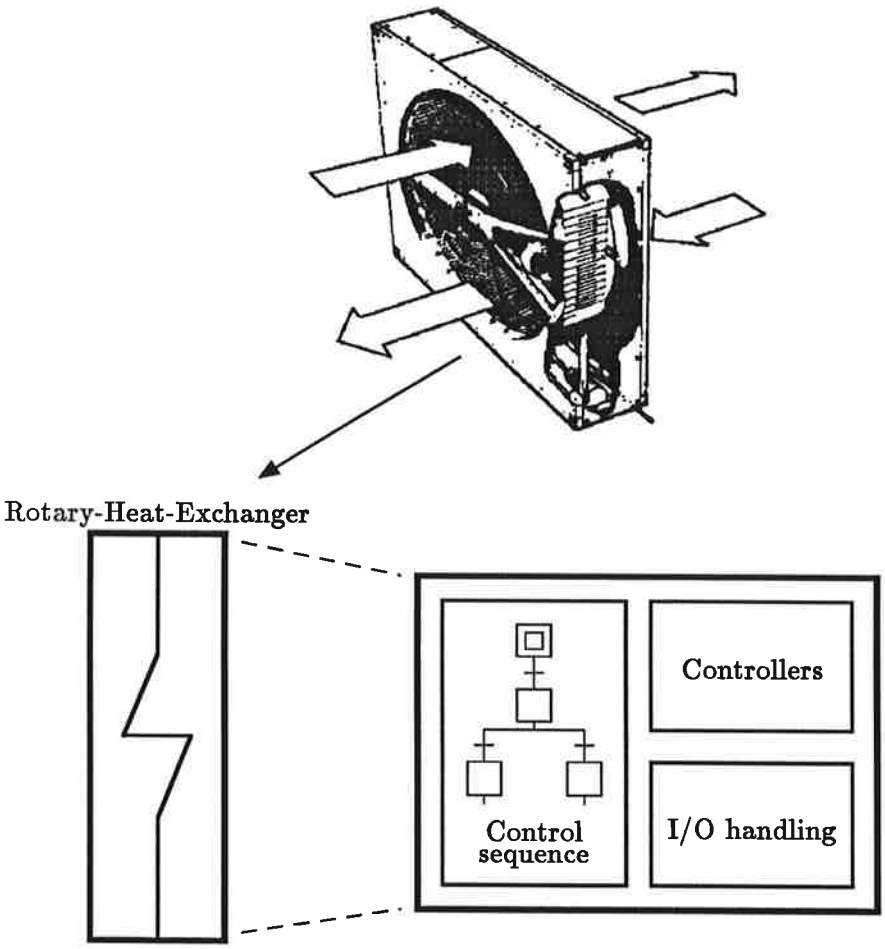


Figure 5.2 The mapping of the physical component rotary regenerative heat exchanger into the class *Rotary-Heat-Exchanger* is illustrated. The control functions associated with the component belong to the component, but they are represented by subobjects to the component object.

Now, what else should the class *Rotary-Heat-Exchanger* contain? As mentioned in the beginning of this chapter, a class can be described as data and operations allowed to be performed on the data. Ideally we therefore would like to gather the control sequence, the controllers and I/O, including AD/DA converters, signal conditioning, alarm handling, etc., and let them be part of the class. This is illustrated in Figure 5.2. The figure shows how a rotary regenerative heat exchanger is identified and then modeled by the class *Rotary-Heat-Exchanger*. In the figure you can see an instance of the class, also called an *object*, which is graphically represented by an icon. The icon looks like the standard symbol for a heat exchanger and is thereby easily recognized in the end-user and control interfaces. The control sequence, the controllers and the I/O belonging to the heat exchanger are also placed within the object.

Component-based structuring applied to the examined system

As pointed out before the component-based structuring leads to a straight forward mapping of physical components into specific classes. When applying this method on the system we have examined you get the tree structure shown

in Figure 5.3. The IPCL application program is divided in a way that makes it possible to isolate the control sequence belonging to each component, and this sequence is then assigned to the corresponding class. For some of the components we find, unfortunately, that it is not possible to isolate the control sequence completely. Since we have three components in the system involved in the controlling of the supply air temperature, it is obvious that they in some way have to communicate with each other. Therefore the class tree in Figure 5.3 might be somewhat treacherous, indicating that the classes *Rotary-Heat-Exchanger*, *Hot-Water-Heating-Coil*, and *Direct-Expansion-Cooling-Coil* are independent of each other, which they are not.

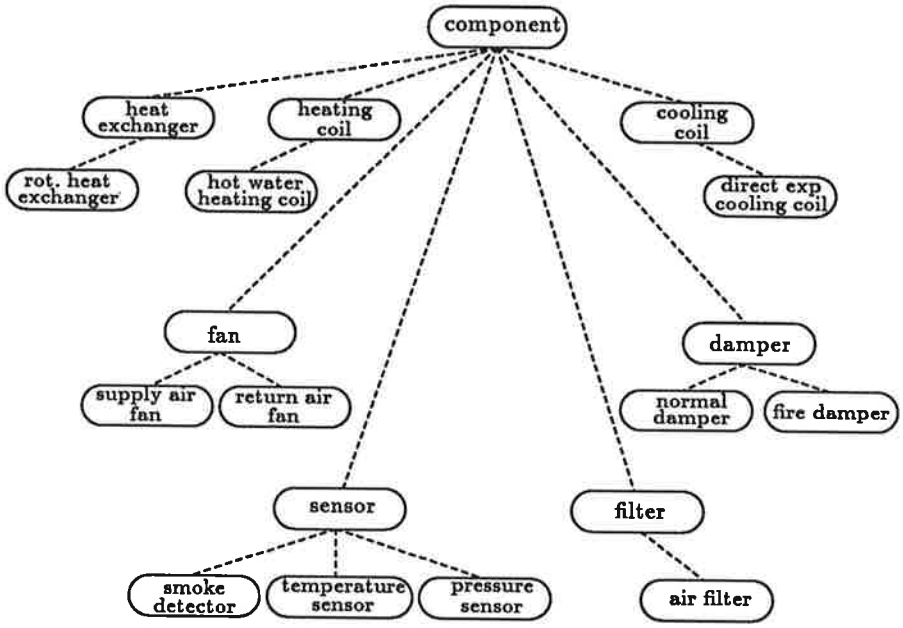


Figure 5.3 The class hierarchy in the component-based structuring.

Each class can be expanded separately whenever there is a need for it, e.g. when another type of heating coil is introduced to the system it simply has to be added as a subclass of *Heating-Coil*. Since the control sequences differ between the supply air fan and the return air fan we have to assign them to different classes. You may ask yourself what other types of filters that are conceivable. There is a great variety of filters and to have the ability of representing the different types with different icons you have to assign them to separate classes. The class *Filter* has no control function assigned.

Naturally, to be able to model the control parts belonging to each class it is also necessary to define classes handling the control sequence, the PID controllers, and the I/O. These classes are not accounted for in this section, since they also are needed in the function-based structuring. You will find a description of them in Section 5.4. In Figure 5.2 we illustrate how the control functions of each component are assigned to the component object. Each function is represented by an object and this object is then located as a subobject to the component object. A component object thus may contain several subobjects and in the case of the rotary heat exchanger in Figure 5.2 we have three subobjects, one describing the control sequence, one modeling the controllers, and another one taking care of the I/O.

Conclusions

Comparing some applications programs you will find that the control sequence, for e.g. the rotary heat exchanger, differs depending on other present components. Naturally the code, for controlling the temperature of the supply air if there are three components affecting the temperature, differs from a case when there is only one component affecting the temperature. Put in another way this means that you, for instance, cannot use the same class *Rotary-Heat-Exchanger* in every system, because the control sequence differs in some way depending on if there is a cooling coil present or not. A possible way to overcome this problem is to define different subclasses of the class *Rotary-Heat-Exchanger*, each one with its specific control sequence. Assume that the rotary heat exchanger can appear in four different combinations together with or without a heating coil and/or a cooling coil. Then you have to define four subclasses of *Rotary-Heat-Exchanger*, and you also will be forced to define different subclasses for each appearance of the heating coil and the cooling coil respectively. This will lead to a high number of classes in the future design library, which might be a bit confusing.

To build the end-user interface this method is perfect due to the straightforward mapping of physical components. You will, with a little help from additional classes such as *Connections* representing air channels in the end-user picture, etc., be able to generate a suitable picture for supervising the plant. The control interface, on the other hand, will not be as easy to survey. For example, the sequence for controlling the supply air temperature will be divided and assigned to three different objects. An advantage is that both the end-user and the control interfaces can be generated at the same time.

5.4 Function-based structuring

In the *function-based structuring* the classes are chosen from the control programs point of view. The application program is now divided into larger parts, where each part is handling a certain function, e.g. controlling the temperature of the supply air. For each of these functions a class is defined and in the supply air case the class *Supply-Air-Temperature-Control* is suitable. This class then contains the control sequence for the supply air temperature control. As in the case of component-based structuring it is also desirable to assign the PID controllers and the I/O-handling, dealing with the function, to the class. This procedure is illustrated in Figure 5.4. The class *Supply-Air-Temperature-Control* is defined from a system where three components control the temperature namely a rotary regenerative heat exchanger, a heating coil and a cooling coil. The look of the icon, representing the class, is here chosen to show the symbols of the physical components. This is of course optional, but we think it is a nice way of informing the user what components the object is controlling.

Function-based structuring applied to the examined system

The major parts of the application program turned out to be sequences controlling the temperature and the pressure of the supply air as well as the pressure of the return air. There are three components involved in controlling the tem-

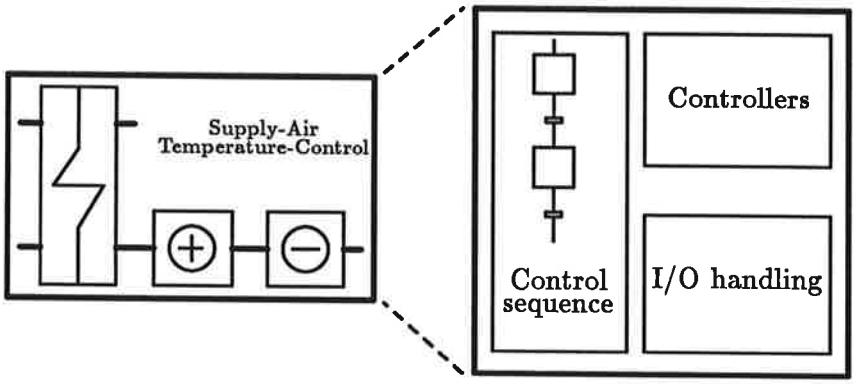


Figure 5.4 The class *Supply-Air-Temperature-Control* is defined and the control sequence, PID controllers and I/O are assigned into the class. There are three physical components controlling the temperature namely a rotary heat exchanger, a heating coil and a cooling coil.

perature namely a rotary heat exchanger, a heating coil and a cooling coil. The pressure control is handled by a supply air fan and a return air fan. Due to the described method we thus define the classes *Supply-Air-Temperature-Control*, *Supply-Air-Pressure-Control* and *Return-Air-Pressure-Control*, cf. Figure 5.5.

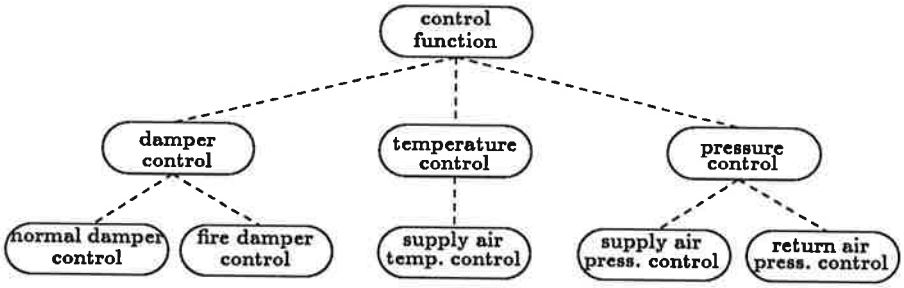


Figure 5.5 The basic classes in the function-based structuring.

The reason for choosing the class *Supply-Air-Temperature-Control* to be a subclass of the class *Temperature-Control* is that you might want to define the class *Return-Air-Temperature-Control* in a system that instead controls the temperature of the return air. To be able to model the control sequence of the different dampers in the system, the classes *Normal-Damper-Control* and *Fire-Damper-Control* are also defined. In order to model the control part of the classes there is a need for additional classes such as *Heat-Exchanger-Grafcet*, *PID-Controller*, *TA-Controller*, *Analog-Inport*, *Analog-Outport*, etc., cf. Figure 5.6. In Chapter 7 these additional classes are described. The class *Hierarchical-Block* is mentioned in Section 6.1. When controlling a "real" AHU there are of course much more details you have to take into consideration and this will increase the number of classes required.

Conclusions

This approach is not applicable when generating the end-user interface but instead the control interface is easily generated. Using this method when structuring will result in a more compact description of the control part of the system which, in our opinion, will be easier to grasp than the one that appear in the component-based structuring. The communication between objects

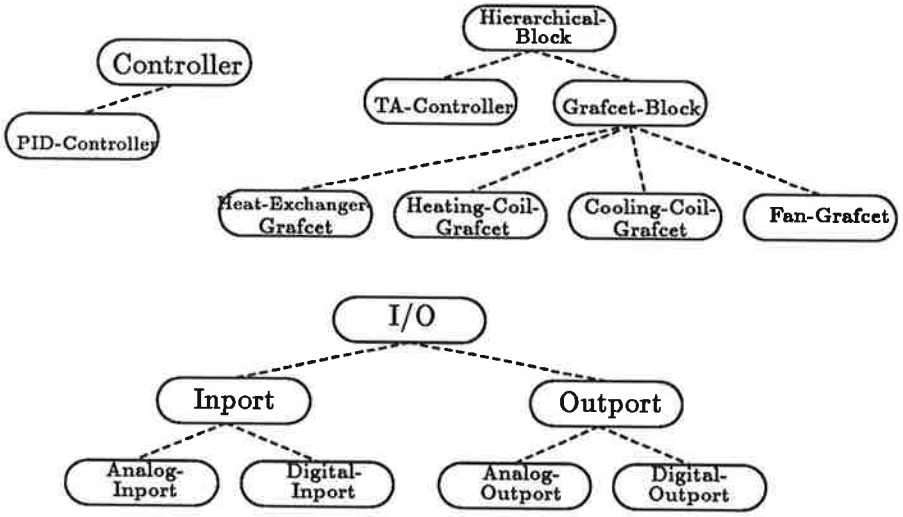


Figure 5.6 Additional classes in the function-based structuring. These classes are also used when describing the control part of the classes in the component-based structuring.

from different classes, involved in the same function, e.g. controlling the supply air temperature, is eliminated using this approach, and thus the sequence concerning the control of the supply air temperature can be gathered in one single object.

One way to overcome the problem with generating the end-user interface is to make sure that when you select, e.g., a *Supply-Air-Temperature-Control* object and place it on the control interface, there will at the same time be created objects upon the end-user interface. These objects should then show how the components involved in controlling the temperature are connected to each other in reality. In our example we get three graphical objects representing the rotary heat exchanger, the heating coil and the cooling coil.

The problem with large future design libraries, discussed in the conclusions of Section 5.3, will with this approach be somewhat reduced. Instead of defining a subclass for each component you now only have to define one subclass of the class *Supply-Air-Temperature-Control* for every possible combination, cf. Figure 5.7. In the figure subclasses have been defined for a rotary heat exchanger controlling the temperature of the supply air together with different combinations of a heating coil and a cooling coil.

5.5 Combined structuring

By combining a modified version of the component-based structuring and the function-based structuring you can utilize the benefits of each approach. The modified version of the component-based structuring is now performed without assigning the control parts to each class. Thus, the classes will provide "empty" objects serving only as icons. The function-based structuring is carried through exactly as described in Section 5.4. The class tree is then obtained by merging the class trees in Figure 5.3 and Figure 5.5 into one, cf. Figure 5.8.

Note that the classes corresponding to component-based structuring do not

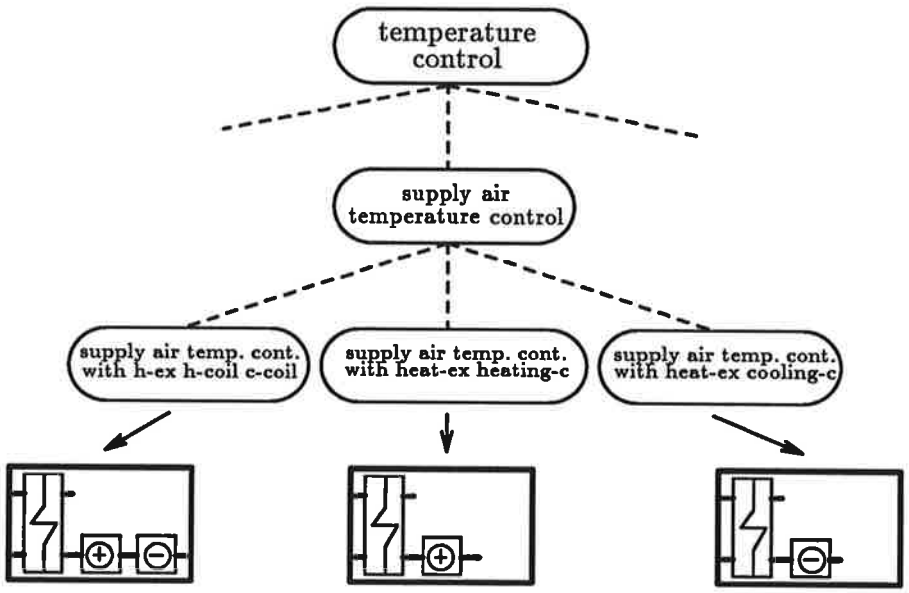


Figure 5.7 Examples of possible subclasses to the class *Supply-Air-Temperature-Control*. For every existing combination of components controlling the supply air temperature, a new subclass must be defined. The figure also shows an object, with a suitable icon here representing the involved components, of every subclass.

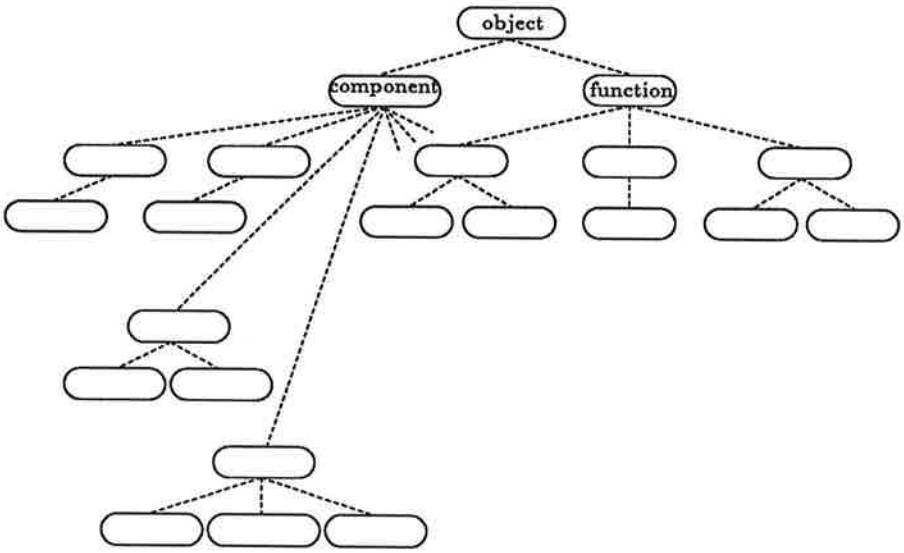


Figure 5.8 The tree structure in the combined structuring. The classes from the component-based structuring are merged with the classes from the function-based structuring. The subclasses of the class *Component* are all "empty" by means of control sequences, PID controllers etc.

include the control sequence, the controllers and the I/O handling. These functions are instead accounted for by the classes corresponding to the function-based structuring. Hence, the end-user interface and the control interface have to be generated separately.

5.6 Summary

We have presented three different ways of structuring, and also pointed out the advantages and disadvantages of each alternative.

Is it possible to carry the component-based structuring through? We only have examined one system thoroughly and in this system we omitted some of the control functions and still some difficulties arose that we think will affect an extensive system even more. A problem might occur in a real time implementation, due to the communication between the different control blocks, since the control program of the different components must exchange information with other components involved in the same control function. As one example the temperature control may involve three components: a heat exchanger, a heating coil and a cooling coil.

As mentioned before a problem with the function-based structuring arises when generating the end-user interface. We therefore find that the best method is the combined structuring, since this approach makes it possible to benefit from both the function-based and the component based structuring.

6. Function blocks and Grafcet

In Chapter 5 we have described how to structure the HVAC system. This results in different objects that, e.g., control the temperature or the pressure, which should be implemented. In the following we will describe how we chose to solve this by using *graphical function blocks*. It is possible to implement every single function within the control program with function blocks. You would need blocks that could add, subtract, compare etc. This leads to a rather complicated graphical picture of the system where you have problems finding out how the different signals propagate. Therefore we find that the sequential part of the control program – which is the major part – is better implemented by means of *Grafcet*. Grafcet has been developed to easily describe a sequential net and is now also an IEC standard, cf. Section 6.2. If one expresses the conditions and actions in Grafcet with a compact notation that is easy to understand even for persons with only a little knowledge of programming, we believe that almost any end user could understand in what way the system works. In the following you will find a description of the function blocks as well as of Grafcet, and how they interact.

6.1 Graphical function blocks

Hartman [2] suggests the use of function blocks, which are predefined blocks that, e.g., takes the average of a number of variables. Instead of writing this program sequence yourself, you just use the function block and only fill in what variables should be averaged. This idea is further developed by French [3], who favours graphical function blocks. They also perform a predefined task, but are used in a graphical design environment and are connected to other function blocks with wires. To a graphical averaging block you would connect the function blocks that represent the variables that are to be averaged to inputs on the graphical block. The values on the inputs are averaged and results in a value on the output. Since we are creating a graphical design tool it is natural for us to use graphical function blocks in our implementation.

The graphical function blocks must communicate with each other and we must make sure that there is a flow of information between them. Therefore each function block has in- and outputs that one can connect with wires in an arbitrary way, as long as you connect an output to an input. Each in- and output has a corresponding attribute in the function block to which the value that is coming in from an input is assigned. The function block processes the data and writes the output data in the attributes corresponding to the outputs. These values are read and sent out on the wires. The way we have implemented this flow of data between the blocks is further described in Appendix B.

The function blocks can also be *hierarchical*, a kind of macros, to improve the readability of the system. A hierarchical block has a sublevel that in turn contains function blocks or other hierarchical blocks. In order to describe a hi-

erarchical block the class *Hierarchical-Block* is introduced. All the hierarchical function blocks are then subclasses to the class *Hierarchical-Block*. To transfer signals that concern a hierarchical block to its sublevel we use *connection posts*. Each connection post is associated with an attribute in the hierarchical block and reads a value from or writes a value to the attribute of the hierarchical function block, cf. Figure 6.1. This value is transmitted through the wires and function blocks on the sublevel in the same way as on the superior level. A hierarchical block can, as mentioned above, of course in turn contain hierarchical blocks, and we can thus create as many sublevels as we find appropriate. However, the improved readability that we achieve by using different levels can also deteriorate again if we use too many sublevels, since we cannot get a general view of the system.

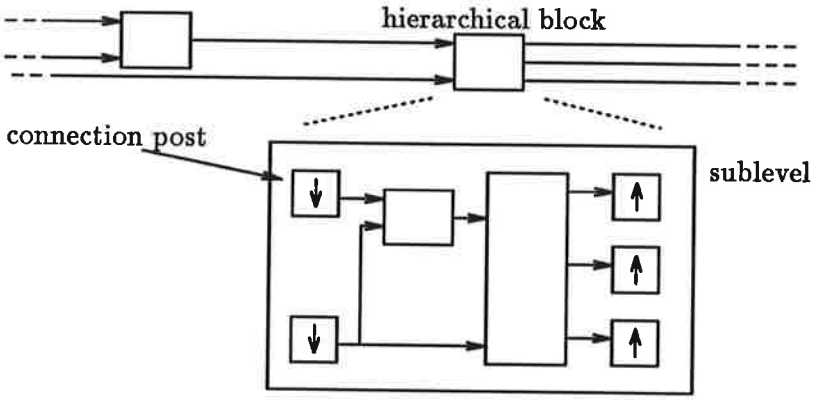


Figure 6.1 A part of a system built from function blocks. Data flows from left to right, and is transmitted to and from the sublevel by connection posts. An arrow pointing down in the connection post means that it transmits data to the sublevel, and an upwards pointing arrow transmits data back to the superior level.

6.2 Grafcet

Grafcet – *Graphe de Commande Etape-Transition* – was developed in France to describe sequential nets. With some small modifications it passed as an IEC standard, IEC 848.

The fundamental building blocks in Grafcet are steps and transitions. A step is always followed by a transition and vice versa, and the first step in a sequence is an initial step. Only one step at a time is active, and the step performs some kind of action – it turns on a heater or something like that. In order to show the user what step is active, a marker is placed in the box representing this step. When you have entered a step the transition that follows is being activated. A transition contains a condition that has to be fulfilled before the sequential program is continued, see Figure 6.2. When this condition is met the transition is fired, and is no longer active. The action of the following step is carried out and a new transition is activated.

You can also have alternative branches in Grafcet that allows you to not always perform the same task in some part of the sequence, Figure 6.3. After a step you place two transitions that both become active at the same time. The

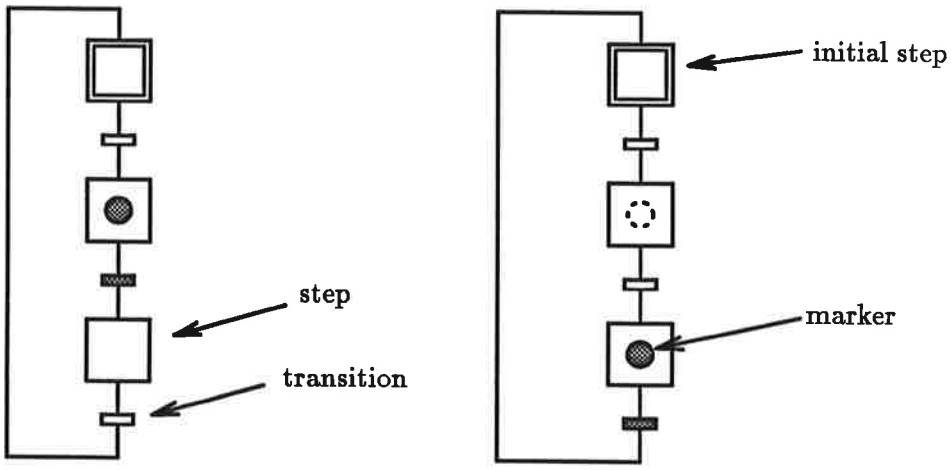


Figure 6.2 A Grafcet sequence. In the right picture the transition that is active in the left one has been fired, and a new transition is activated. The marker shows where the program is executing.

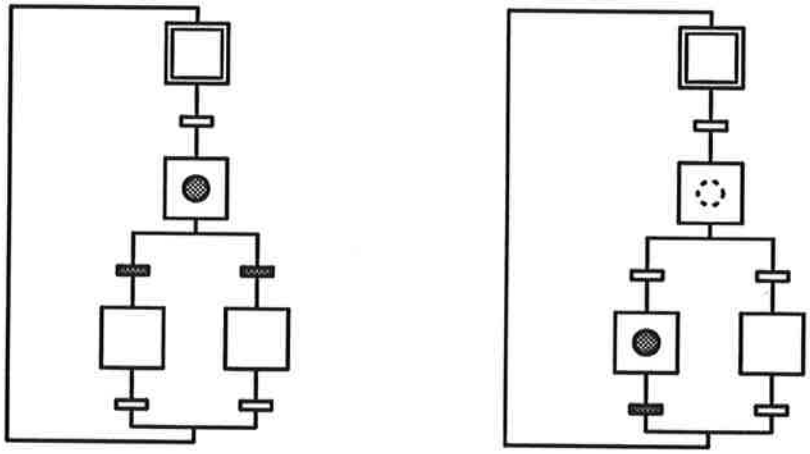


Figure 6.3 The transition that first is fired decides which branch is going to be executed this time.

transition that is fired first decides in which branch the program continues[9].

Another feature involving more than one branch is parallel branches, Figure 6.4. The different branches are executed in parallel, and at the end of the branches the program waits until all of them have finished executing before it continues.

Sometimes the Grafcet sequence becomes too complex. To avoid this you can use macro steps. A macro step contains a part of the Grafcet sequence that is hidden on a sublevel. At the superior level you can only see that the program is executing the macro step, but if you examine the sublevel you get more precise information of what the sequence does.

A non-standard feature that is very useful is the exception transition, Figure 6.5. An exception transition is associated with a macro step, and if the

9 Standard Grafcet requires the transition conditions to be mutually exclusive, but in the implementation in G2 that we have used one of the branches is randomly chosen if both transition conditions are fulfilled simultaneously.

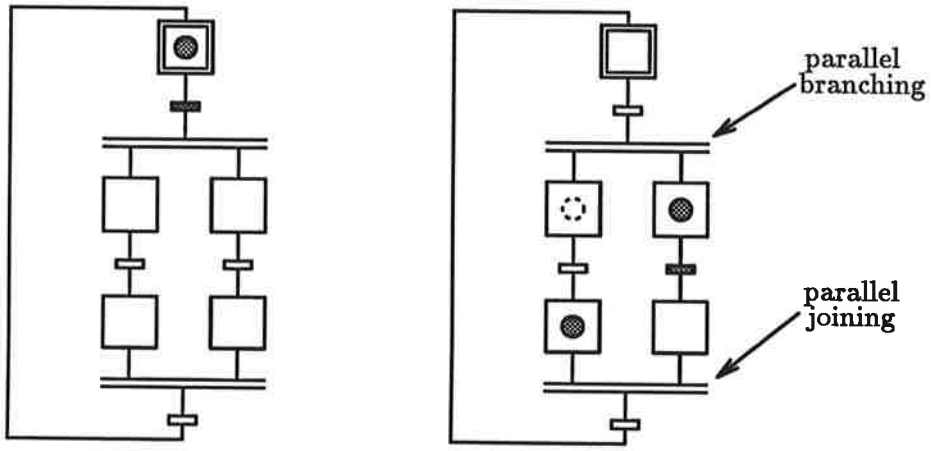


Figure 6.4 The two branches are executed in parallel. The transition after the parallel joining is active only when both of the branches are ready.

condition in this transition is fulfilled, the program leaves the macro step no matter where it is executing, to continue in the exception branch. This is a very neat possibility to use for emergency situations that occur very seldom, but must have an immediate impact on the control sequence.

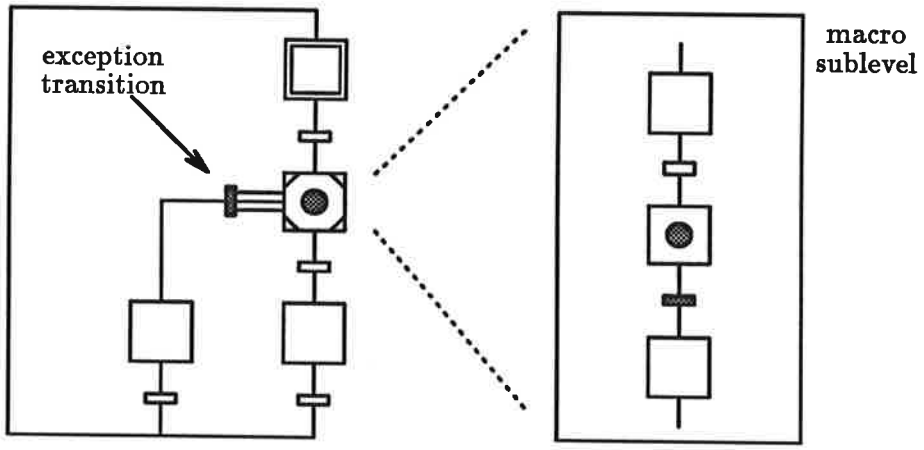


Figure 6.5 The transition after the macro step is not active until the marker has entered the last step. The exception transition is active as soon as the program enters the macro step.

A more complete description of Grafcet is found in the book *Petri Nets & Grafcet* [6].

Grafcet rules

When you use Grafcet for sequence control of an installation, you must make sure that the program will perform in the same way no matter what implementation or version of Grafcet you use. We therefore need a set of rules to formalize the behaviour of a Grafcet sequence. We have the following rules:

1. The initialization activates the initial step.
2. A transition is *fireable* if:

(a) The step preceding the transition is active. In the case of parallel joining all of the steps immediately preceding the joining have to be active, cf. Figure 6.4.
(b) The corresponding transition condition is true.

3. A fireable transition must be fired.
4. When a transition is fired, all the steps preceding the transition are deactivated and all the steps following the transition are activated.
5. All fireable transitions are fired simultaneously.
6. When a step must be both deactivated and activated it remains active without interrupt.

6.3 Grafcet in graphical function blocks

Since Grafcet performs the sequential part of the control, it operates on data and other information from the system. This means that it must be connected to the surrounding world in some way. We take care of this by using a hierarchical function block. The function block containing a Grafcet sequence will look just like any other hierarchical function block. The difference, however, is that on opening up this block you find a Grafcet sequence instead of other function blocks.

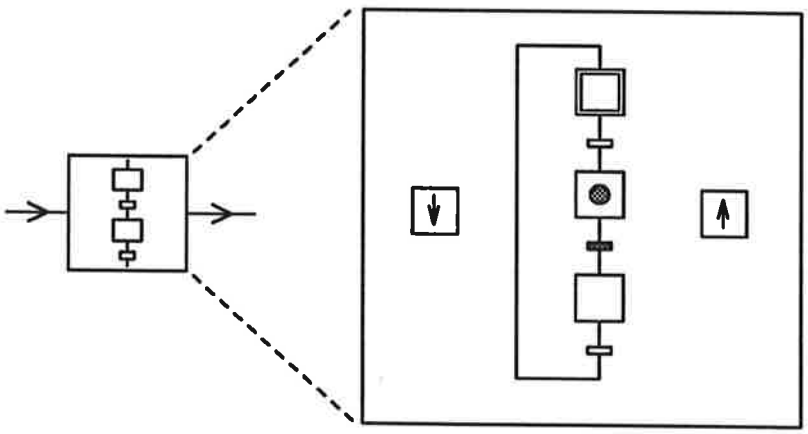


Figure 6.6 This picture shows an object, of a subclass to the class *Grafcet-block*, to the left and to the right the sublevel of the object is shown. Data is transmitted to and from the sublevel by the connection posts.

In Figure 6.6 the hierarchical function block containing a Grafcet sequence is seen to the left and to the right the sublevel of the block is shown. In order to transmit data to and from the sublevel of the Grafcet block we use connection posts. They are not connected to any part of the Grafcet sequence and the reason for this is that the information can be used in more than one step or transition, and that output data also can be produced in more than one of the steps. The icon of the connection post that transmits data to the sublevel has an arrow pointing downwards and the arrow of the connection post that returns output data to the superior level points upwards. These connection posts can thus be treated as parameters whose values are available to any step or transition in the Grafcet sequence by just referring to the name stated

beside the connection post. The name of the connection post is the same as one of the attribute names in the hierarchical block, and if it is an in-connection post the value of the corresponding attribute is read and used as the value of the connection post. The value of an out-connection post is assigned to the corresponding attribute of the hierarchical block. There is a more detailed description of the connection posts in Section 7.1.

Language for Grafcet actions and conditions

We have used a version of Grafcet that is implemented in G2, where all actions and conditions are expressed with the G2 language. This is a language that tries to resemble ordinary English and thus becomes rather tiresome and ineffective. To express that a parameter called enable-hc is to be designated a certain value, say false, you must write *'initially conclude that enable-hc is false'*, cf. Figure 6.7.

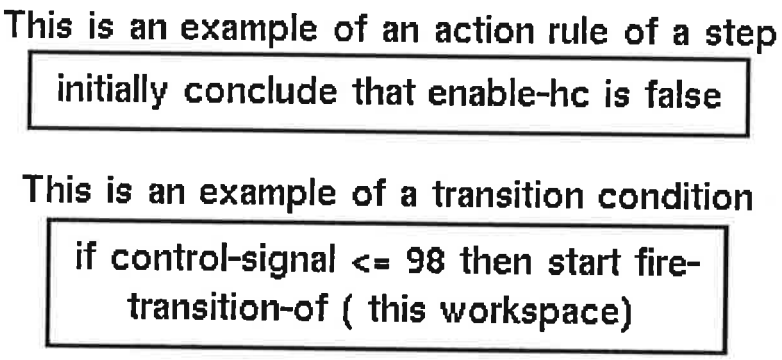


Figure 6.7 This is an example of a Grafcet action and a transition condition written using G2 syntax.

A compact and short notation that serves just as well is e.g.: *'enable-hc = false'*. Likewise a condition can be expressed as *'control-signal <= 98'* meaning *'if control-signal <= 98 then fire this transition'*. A benefit of a compact notation is that the action or condition could be shown on the screen beside the step or transition. A person that wants to debug or watch the program while it runs, easily understands what conditions are fulfilled at a certain time.

One step can perform more than one action as you can see in Figure 6.8. The Grafcet sequence in the figure controls a fan with variable guide vanes and two different speeds: slow and fast. The guide vanes are controlled by a PID controller and when they have reached either a minimum or a maximum the speed has to be altered. This and all other Grafcet sequences are described in Chapter 8.

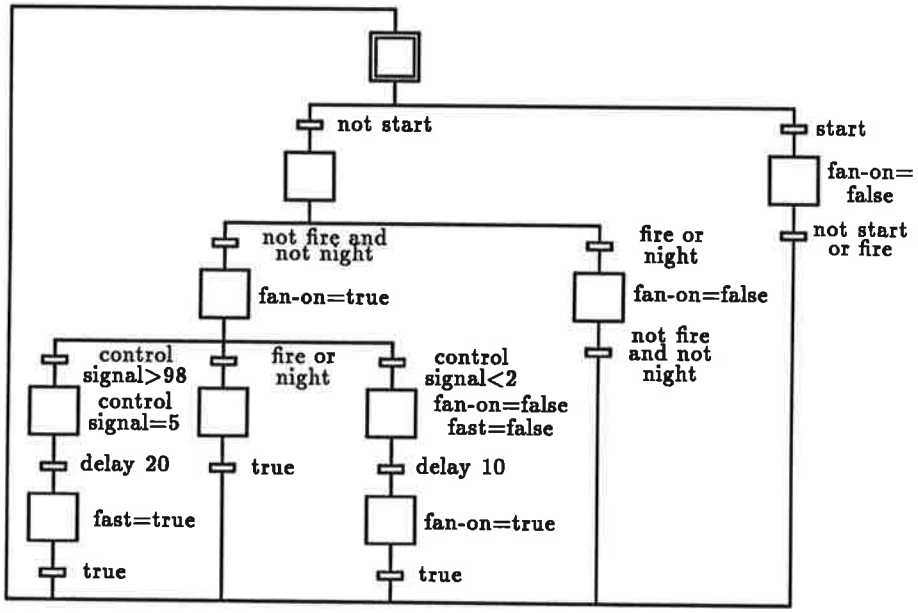


Figure 6.8 This is a Grafcet sequence for controlling a fan in an HVAC installation. To the right of every step and transition is the action or transition condition. The steps without any text do not perform any action at all. If a condition is 'true' then this transition is fireable as soon as it is activated.

7. Implementing a Building Management System using G2

In Chapter 5 we have discussed different approaches to the structuring of an HVAC system. The easiest and most straightforward approach seems to be a mixing of the component based and the function based structuring, the combined structuring. The function based structuring is used when configurating the control program and the component based when creating the dynamical end user interface, and this is also the approach that we have chosen to work with.

It is easy to propose new ideas concerning the structuring of a building management system, but there may turn up unforeseen difficulties when trying to implement those ideas. In order to discover these problems and at the same time make sure that things work as we intended, we have implemented the BMS functions for one specific AHU. TA supplied us with the line program that controls the air-handling unit in their office building and we made an object-oriented structuring of this program. When implementing the BMS we used G2 from Gensym Corp.[1], that supports object-oriented programming and simulation. To verify our program we simulated the air-handling unit using a simple dynamic model described in Appendix C.

Since we cannot make a listing of the implementation in G2 we will try to describe it and the classes we have used to make the control program for the air-handling unit. We have excluded the classes that were needed due to G2 specific reasons in this presentation. During our work with the control program we have sometimes considered different approaches to problems and sometimes chosen not to implement some classes that are essential when actually controlling a real air-handling unit, e.g. classes to describe alarm handling, etc. In these cases we describe the alternatives and explain the reasons for choosing one of them or why we have omitted some classes.

7.1 A description of the different classes

The control functions, identified according to the function-based structuring described in Section 5.4, needed to be able to model the examined system are, cf. Figure 5.5:

- supply air temperature control
- supply air pressure control
- return air pressure control
- normal damper control
- fire damper control

These control functions are implemented using hierarchical function blocks as described in Section 6.1, and thus they are subclasses of the class *Hierarchical-Block*. We only describe the *Supply-Air-Temperature-Control* block since it contains objects from all the essential classes. In the figures, that are screen dumps from G2, the heat exchanger, heating coil, and cooling coil will be abbreviated he, hc, and cc. Figure 7.1 shows the hierarchical function block controlling the supply air temperature. This block is is an instance, i.e. an object, of the class *Supply-Air-Temperature-Control*. Figure 7.1 also shows the objects that connect the *Supply-Air-Temperature-Control* object to the surrounding world. There is one object that reads the temperature setpoint value, one inport block, and four outport blocks. These objects belong to the classes *Get*, *Inport*, and *Outport* and will be described in the following text.

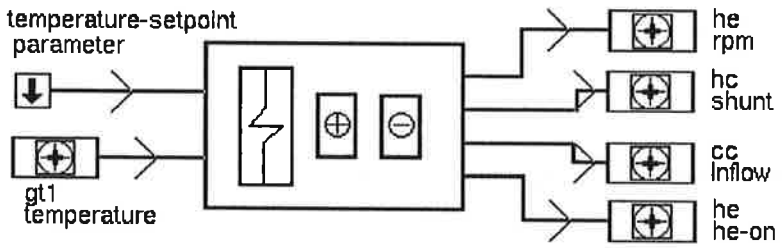


Figure 7.1 The *Supply-Air-Temperature-Control* object with the objects that connect it to the surrounding world.

The class Get

The object that reads the setpoint is an object of the class *Get* and its icon is a square with a downwards pointing arrow. It can read a value from any attribute within another object and the first label refers to that object's name and the second one to the attribute. In this case it fetches the value from a parameter (a standard G2 item, cf. Appendix A) whose name is temperature-setpoint, cf. Figure 7.2.

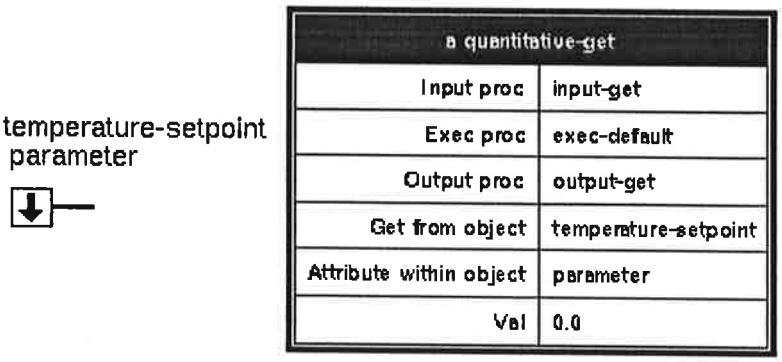


Figure 7.2 To the left you can see the object of the class *Get*, and to the right there is a table of the attributes. The upper three attributes state the input, execute and output procedures. The next two attributes refer to where the object fetches the value to put in the attribute val. This value is then transmitted out to the connecting wire by the output procedure.

The classes Inport and Outport

The inports and the outports connect the control program to the world outside the RPU. To be able to handle analog and digital signals we have to define two subclasses to the classes *Inport* and *Outport*. Thus we work with four different classes: *Analog-Inport*, *Digital-Inport*, *Analog-Outport*, and *Digital-Outport*. The only difference between the analog and the digital ports is that the attribute is of type quantity or logical. Another way of defining classes would be to define the classes *Analog-Port* and *Digital-Port*, and then defining two subclasses to each of them, one class for an inport and another one for an outport. Since we simulate the air-handling unit in G2 we do not use physical ports, but instead indicate what signal from the plant that should be present at a certain port. As in the case of Get the first label indicates which physical component the port is connected to and the second one which signal from this component, cf. Figure 7.1.

In a real system the in- and output signals must pass through an I/O block, i.e. an analog or digital in- or output block, that should include a filtering block. The reason for this is of course that you may want to filter the signal but also that it sometimes is necessary to scale the signal and remove a possible bias. Besides this, some kind of alarm handling block is indispensable in a real system, but we have not implemented any alarm handling features and therefore have not dealt with anything concerning alarms. If the functions mentioned above were to be implemented, it would be convenient to locate the objects representing these functions on a sublevel to the objects from the classes *Inport* and *Outport*.

Continuing down on the sublevel of the *Supply-Air-Temperature-Control* block we find the objects present in Figure 7.3. The setpoint and the measured temperature are transmitted to the three controllers that control the heat exchanger, heating coil, and cooling coil. There are signals coming from the *Grafcet-Block* objects, in the bottom of the figure, that determine if the controller output will be decided by the controller itself or if it should be forced to a certain value. The *Grafcet-Block* objects handle the sequential control of the heat exchanger, heating coil, and cooling coil and they need data from each other and from the respective controller output.

The classes In-Connection-Post and Out-Connection-Post

These two classes are both subclasses to the class *Connection-Post*. To the left in Figure 7.3 there are two *In-Connection-Posts* with downwards pointing arrows that receive data from the superior level in Figure 7.1 down to this sublevel. To the right there are four *Out-Connection-Posts* that transmit data back up to the superior level, and their arrows point upwards. The connection posts have a corresponding attribute in the superior function block, and read or write data in this attribute, cf. Figure 7.4. The name of the associated attribute is shown next to the post. Of course you can receive and transmit either a quantity or a logical value with the connection posts and thus you need different classes for quantity and logical connection posts. In the figure the *Out-Connection-Post* labeled *he-on* transmits a logical value.

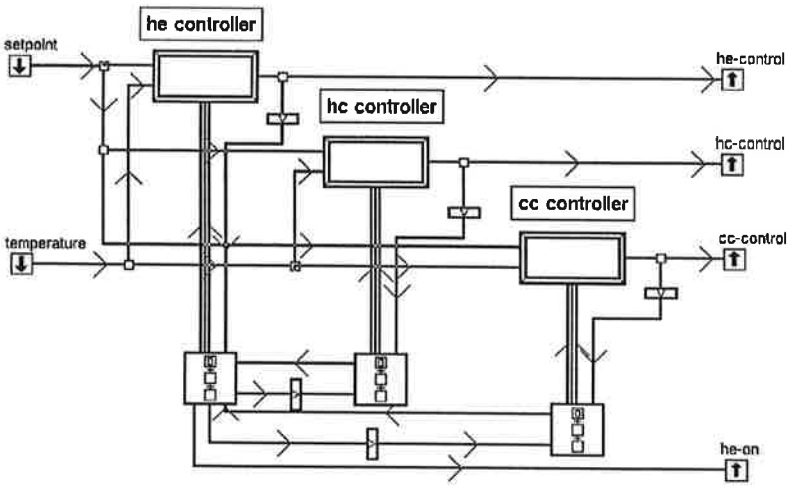
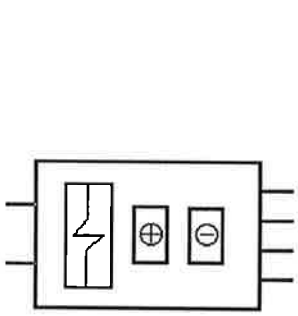


Figure 7.3 The controlling of the heat exchanger, heating coil and cooling coil are handled by *Grafcet-Blocks* and *TA-Controllers*. These objects are all located at the sublevel of the *Supply-Air-Temperature-Control* object.



a vvx-vb-kb-control	
Input proc	input-vvx-vb-kb-control
Exec proc	exec-hierarchical-block
Output proc	output-vvx-vb-kb-control
Setpoint	0.0
Temperature	0.0
He control	0.0
Hc control	0.0
Cc control	0.0
He on	false

Figure 7.4 The attributes in the *Supply-Air-Temperature-Control* block have corresponding connection posts on the sublevel, cf. Figure 7.3.

The class TA-Controller

The rectangulars under the labels *he controller*, *hc controller*, and *cc controller* in Figure 7.3 are objects from the class *TA-Controller*. This class is a subclass of the class *Hierarchical-Block*, and on the sublevel we find a *PID-Controller* and a *Logical-Block*, shown in Figure 7.5. We have called the class *TA-Controller* since it resembles the controllers that are used today in the RPU of the TA SYSTEM 7, where you can force the controller output to any value between a minimum and a maximum value. The *TA-Controller* is an ordinary PID controller and the control signal then enters the logical block where it either may pass or some forced value is sent out instead, depending on the status of this block. The *Logical-Block* as well as the *PID-Controller* are not hierarchical ones, so they do not have an underlying structure. Instead

there are procedures, cf. Appendix B, that take care of the logic and the PID algorithm.

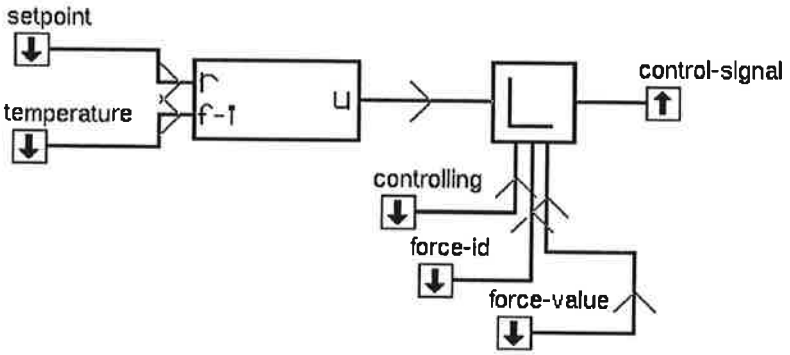


Figure 7.5 The contents of a *TA-Controller* is a normal *PID-Controller* and a *Logical-Block*. The signal *controlling* is true when the output from the controller is to pass unhindered. If not, the *force-id* decides whether the minimum, the maximum or the *force-value* is to be sent out from the TA-Controller.

The class Grafcet-Block

The square blocks in the lower part of Figure 7.3 all contain Grafcet sequences that sequentially control the heat exchanger, heating coil and cooling coil. The blocks are hierarchical and thus objects from different subclasses to the class *Hierarchical-Block*, cf. Figure 5.6. Opening them the steps and transitions familiar from Section 6.2 are found. We have chosen to connect the *Grafcet-Blocks* with other function blocks by wires, but another possibility would be to refer to parameters that are defined at the level superior to the *Grafcet-Block*. The reason for connecting the *Grafcet-Block* to other function blocks with wires is to show how it is related to other objects and what parameters it affects. Yet another reason is that if you use a global parameter you loose the overview of the flow of information in the system. One reason for not using wires but instead refer to global parameters is that there is no object in the Grafcet sequence to connect the incoming wires to, cf. Figure 7.6. As seen in the figure, the Grafcet sequence needs information about the heating and cooling coils and produces signals that control the heat exchanger, heating and cooling coils. These in- and outsignals are transferred with a special subclass of *Connection-Post* that do not have any incoming or outgoing wires. They are thus global parameters on the level of the Grafcet sequence and can be referred to by naming the attribute that they are associated with and that is stated next to the connection posts. In every other sense they are similar to the connection posts described before. All the control sequences implemented in Grafcet are described in detail in Chapter 8.

As you can see in Figure 7.3 we have three different Grafcet blocks for the heat exchanger, heating coil and cooling coil. Of course it is possible to make only one Grafcet sequence that control all three components, but it would probably become a rather complex net that is quite difficult to survey. When supervising the system it is much easier to be able to watch the behaviour of one component at the time. On the other hand we would not have to deal with the communication between the Grafcets that we have in our example. This is all very much a matter of taste and both solutions are equally satisfying.

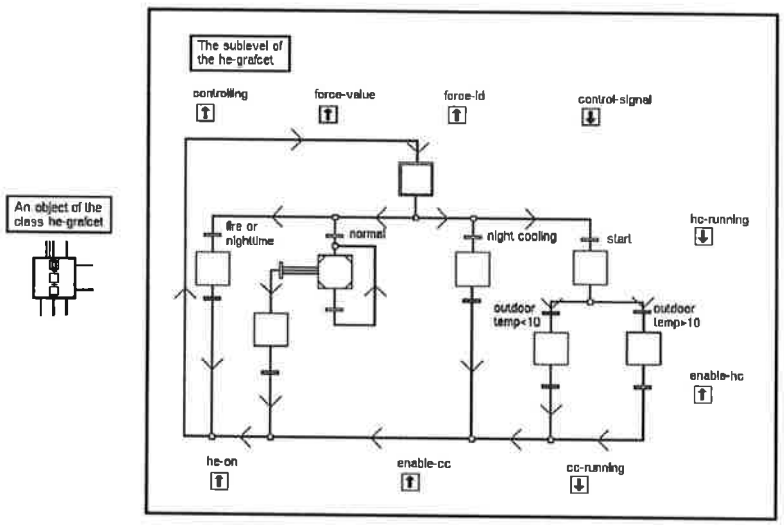


Figure 7.6 An object of the class *Heat-Exchanger-Grafcet* and its sublevel. Note that there is one connection post corresponding to each wire in to or out from the *Heat-Exchanger-Grafcet* block that receives and transmits data to and from the sublevel.

The class Loop-Block

In Figure 7.3 you find small rectangular function blocks with the symbol >. These blocks break open a loop that otherwise will make it impossible to decide in what order the function blocks should be executed, cf. Appendix B. In Figure 7.3 the *Grafcet-Block* object needs information from the *TA-Controller* object and the *TA-Controller* object needs information from the *Grafcet-Block* object. There is no way to determine an execution order of these two function blocks.

The other classes

Since the rest of the function blocks that deal with the control of the air-handling unit, have the same structure as the *Supply-Air-Temperature-Control* block described in the previous section, no additional classes are needed. However, there are some further classes that are necessary when making the end-user interface. Each physical component must also have a corresponding class, cf. Figure 5.3.

7.2 Automatic renaming of Connection-Posts

As mentioned above, in Grafcet actions and transitions, a connection post is referred to by stating its name. This name must be the same as the corresponding attribute in the superior hierarchical block. When implementing this in G2 one unforeseen problem appeared. All names in G2 are global and if for example more than one fan is to be controlled using identical control blocks suddenly two connection posts in the system will have identical names – one in each *pressure-control* block. To remedy this the connection posts must have unique names but this implies that the attributes in the hierarchical blocks

also must have different names and the whole point of object-oriented programming is lost.

The solution to this dilemma is to let the connection posts have identical names when configuring the system. When starting the program an initialization procedure searches for connection posts and renames them by adding a unique number to the name. If there are two connection posts in the system that are named 'fan-on' they will be renamed 'fan-on-1' and 'fan-on-2'. When renaming the connection posts, however, it is not necessary to have numbers in consecutive order for posts with identical names. It is sufficient to go through the posts in the order that they are found in the system and have a common numbering for all the connection posts in the system. This means that although there might only exists two posts that are named 'fan-on' they may be renamed 'fan-on-1' and 'fan-on-12'. The post and its new name are related to each other using the relation feature in G2. The reason for this is that after renaming the connection posts all actions and transition conditions in the Grafcet sequence referring to a certain post must be rewritten to refer to its new unique name. All connection posts are once again gone through and each step and transition on the same level, and on possible sublevels (i.e. in macro steps), as a certain post are examined to see if they refer to this post. If so the name is substituted with the unique name that is related to the post. All changes that are made dynamically are valid only as long as the program runs. When it stops the names, actions and conditions will be changed to their original form.

8. Grafcet sequences in the Building Management System

In this chapter the Grafcet sequences that we have implemented are described in order to show how the sequential control can be handled. The different sequences are the pressure control, the temperature control, the damper control, and the operation modes. In the figures below, the Grafcet transition conditions and step actions are not expressed according to the suggestion in Section 6.3, but instead short remarks are made in the picture. All figures in this chapter are extracted from our implementation in G2 of the examined AHU.

8.1 The pressure control Grafcet sequences

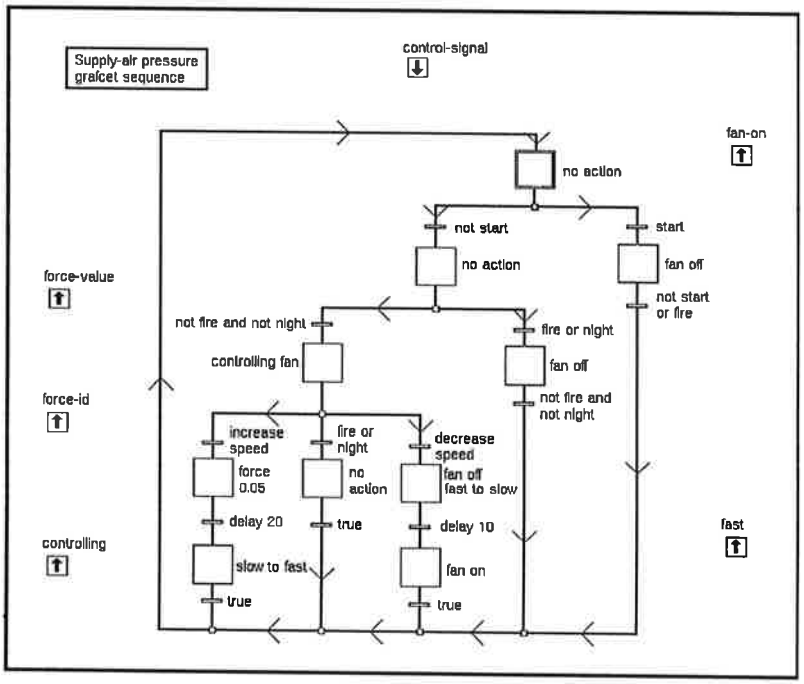


Figure 8.1 The Grafcet sequence for controlling the pressure of the supply air.

The Grafcet sequence that controls the supply-air pressure, cf. Figure 8.1, is controlling a fan with two different speeds, and variable guide vanes that are controlled by the *control-signal*. When the AHU starts, e.g. in the mornings, the fan is turned off as in the right main branch of the Grafcet sequence. If the operation mode is not start but instead fire or night the fan is also turned off. In the other operation modes the fan is to be controlled as seen to the

left of the figure. If the guide vanes are fully open and the fan runs at low speed, the sequence continues in the far left branch and the control signal is forced to the value 0.05. After a delay the speed is changed from slow to fast and the fan is once again controlled normally. If, on the other hand, the guide vanes are closed and the fan is running at high speed, the fan is turned off and the speed is changed to slow. After a delay the fan can be turned on again and controlled normally. The operation mode can of course be changed when controlling the fan and therefore there is an escape branch that is chosen when the mode becomes fire or night.

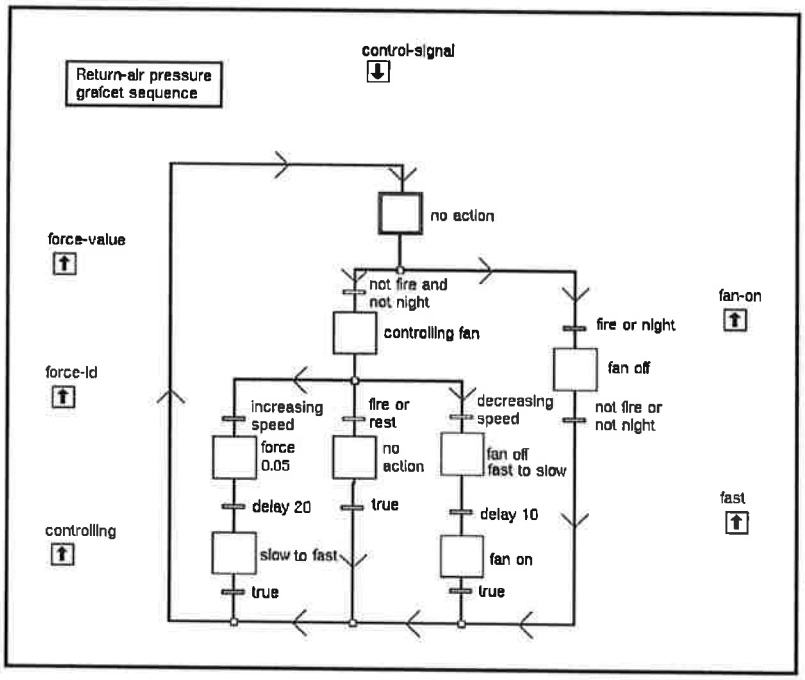


Figure 8.2 The Grafcet sequence for controlling the pressure of the return air.

Figure 8.2 shows the Grafcet sequence for controlling the return-air pressure. The only difference from the sequence above is that there is no special tasks to perform when starting the AHU.

8.2 The temperature control Grafcet sequences

The temperature is controlled by three components: the heat exchanger, the heating coil, and the cooling coil. It is of course possible to merge the Grafcet sequences of these three components into one, but we have chosen to keep them divided in three parts, even though this makes it necessary to communicate between the three sequences. This split up version must be used when making a component based structuring.

The heat exchanger Grafcet sequence

When going through this section the reader should keep in mind that the heat exchanger is the heart of the temperature control. The heating coil and cooling

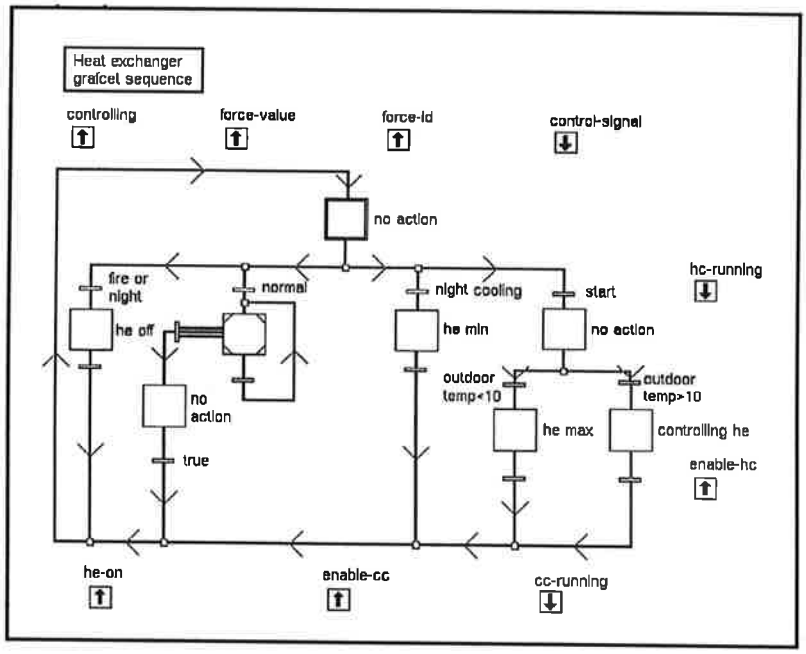


Figure 8.3 The Grafcet sequence for controlling the heat exchanger.

coil are only to be active when the heat exchanger is incapable of controlling the temperature by itself.

As seen in Figure 8.3 the heat exchanger is turned off if the operation mode is fire or night. In normal mode the heat exchanger is controlled by the macro step, which is described below. The macro step can be left even though the operation mode still is normal, hence the loop back into the step. In case the mode is changed the exception transition is used to escape the macro step. During night it is possible to lower the temperature in the building by circulating cold outside air in the building, in which case the heat exchanger is run at minimum speed. When the AHU is started the heat exchanger is run at maximum speed if the outdoor temperature is below ten degrees Celsius, otherwise it is controlled normally.

In Figure 8.4 the macro step mentioned above is exposed. If the air in the building has a lower temperature than the outside air, the cool outlet air can be used to chill the supply air, and the heat exchanger is run at maximum speed. When this condition is no longer met the heat exchanger is turned off for a while. The Grafcet sequence loops back into this macro step again and the cooling coil is disabled. If the heating coil is running, the heat exchanger is to run at maximum speed. When the heating coil is not running but the heat exchanger is running at maximum speed the heat exchanger is to be controlled normally and the heating coil is to be enabled. The other alternative is that the heating coil is not running and the heat exchanger is not running at maximum velocity, and now the heating coil is disabled. If, furthermore, the heat exchanger is not running at minimum velocity it is to be controlled normally. Otherwise the cooling coil is enabled. When the cooling coil is running, the heat exchanger is to run at minimum speed, otherwise it is to be controlled normally. The transition conditions that allow the sequence to reach the exit step of the macro are all written so that they are fulfilled as soon as the conditions that led the sequence into the branch are no longer met. The sequence

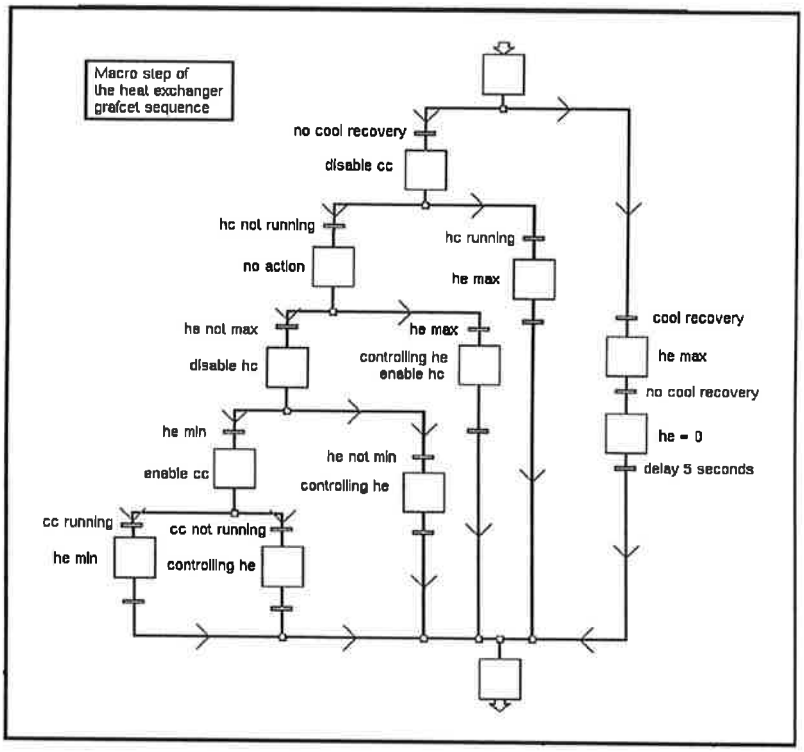


Figure 8.4 The macro step of the heat exchanger Grafcet sequence.

loops into the macro step at the top and another branch is chosen.

The heating coil Grafcet sequence

The first thing that is done in this sequence, cf. Figure 8.5, is that the value of *hc-running* is set to false. The same choices of branches are made as in the case of the heat exchanger. During night, in case of fire, or when the inside air is to be cooled of during night the action of the heating coil is minimized. When starting the system the coil is running at maximum if the outdoor temperature is below ten degrees Celsius and otherwise controlled normally. The macro step in normal mode is described below, cf. Figure 8.6.

If the air is to be cooled off by recovering chill from the return air, the heating coil is to run at minimum. Otherwise, if the coil is disabled it runs at minimum, but has the coil been enabled the heating coil is controlled normally. In case the heating coil still does not run this is reported by setting the *hc-running* to false. If, on the other hand, it is running the *hc-running* is set to true.

The cooling coil Grafcet sequence

By comparing the Grafcet sequence in Figure 8.7 with Figure 8.5 the only difference that is detected between the heating coil and the cooling coil is that during start of the system the cooling coil is always running at minimum. The layout of the macro step in normal mode, cf. Figure 8.8, is identical to that of Figure 8.6 and therefore needs no further explanation.

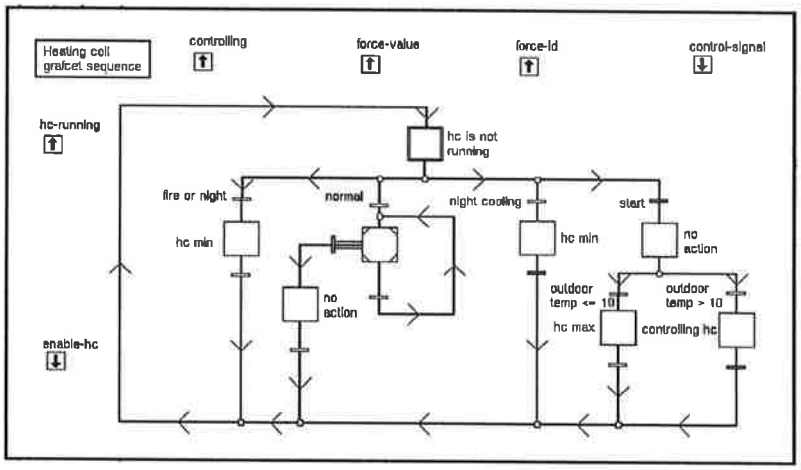


Figure 8.5 The Grafcet sequence of the heating coil.

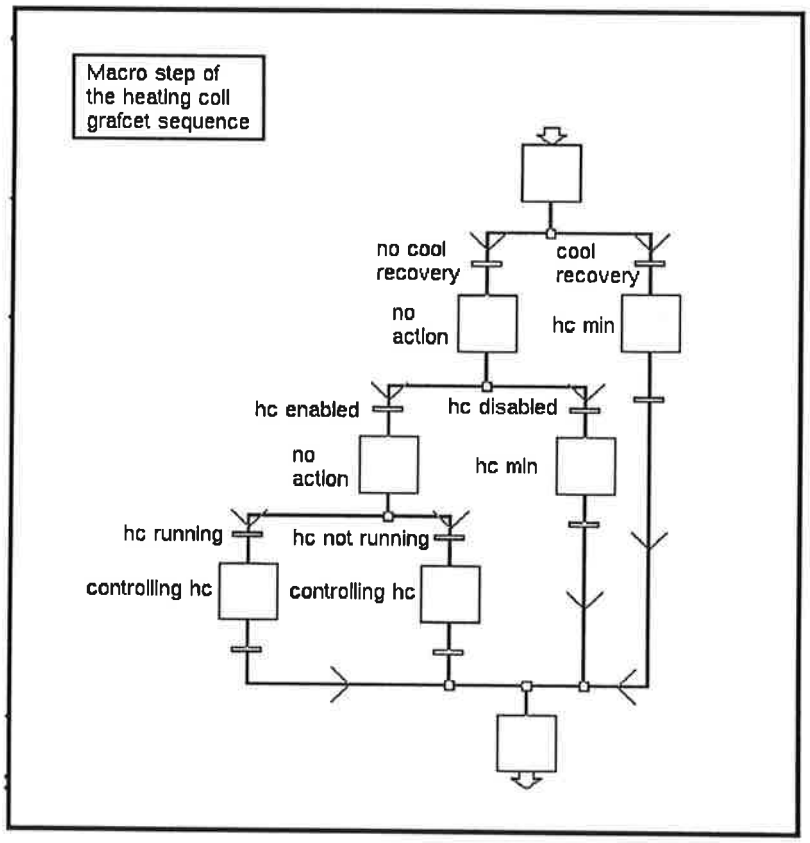


Figure 8.6 The macro step of the heating coil Grafcet sequence.

8.3 The damper control Grafcet sequences

The Grafcet sequence in Figure 8.9 refers to a damper that lets air in or out of the building and that is closed in fire or night mode and otherwise open. A fire damper is open when in fire mode and closed otherwise, cf. Figure 8.10.

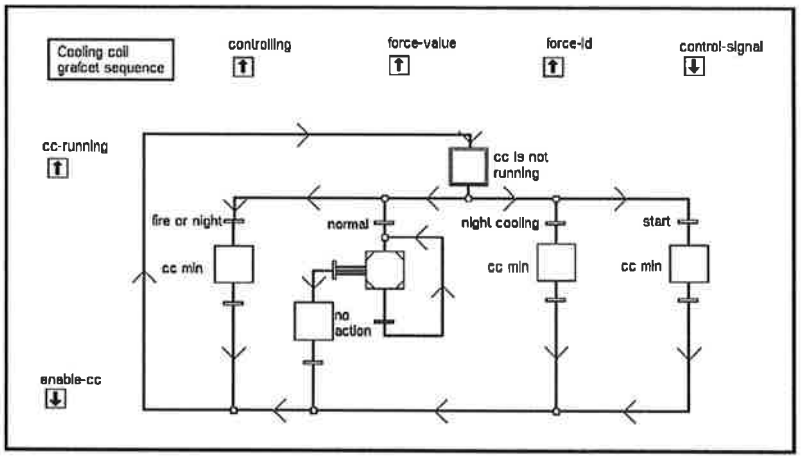


Figure 8.7 The Grafcet sequence of the cooling coil.

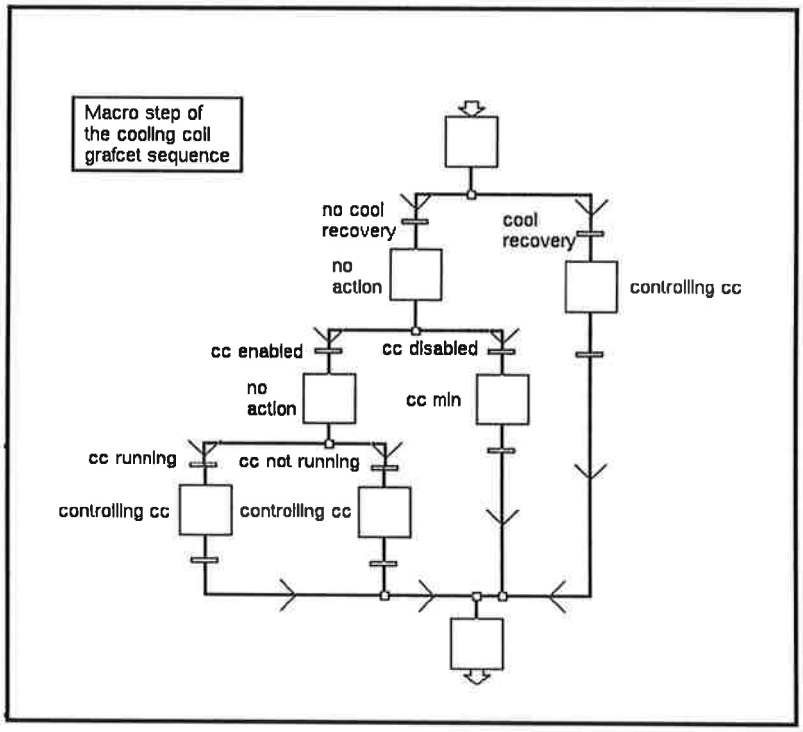


Figure 8.8 The macro step of the cooling coil Grafcet sequence.

8.4 The operation modes

The different operation modes in the system, e.g. day or night operation, must be selected with the help of a clock. Therefore we need a class that can handle time, and signal when it is time to turn the plant on or off, etc. When simulating this system we did not deal with this but instead used buttons to go through the operation modes of the system. Every mode corresponds to a logical parameter that is true when this mode is to be activated and false otherwise. A Grafcet sequence can be used when deciding the operation mode, cf. Figure 8.11. The transition conditions will have to check the time or logical parameters that affect the operation mode. Since the fire mode always must

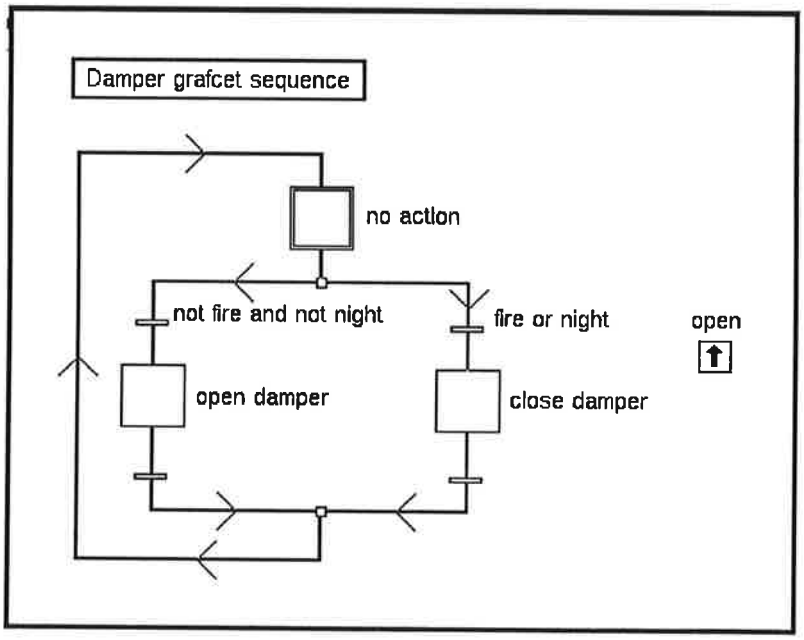


Figure 8.9 The Grafcet sequence for controlling a normal damper.

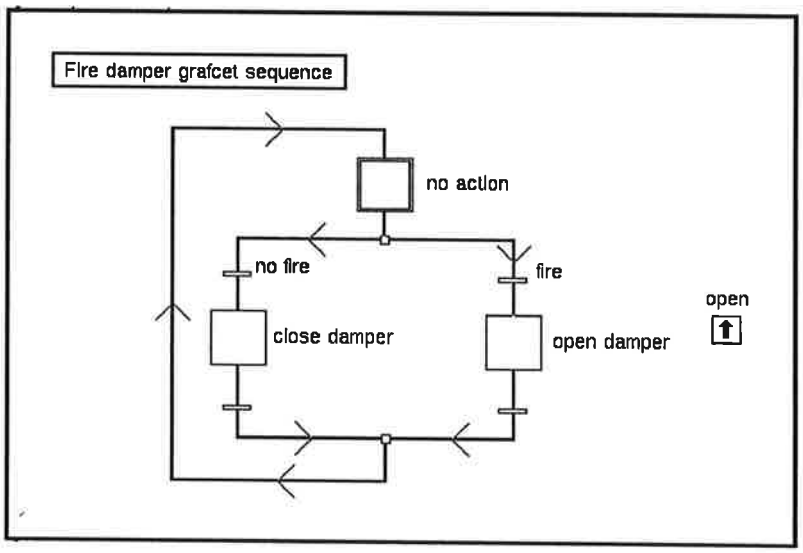


Figure 8.10 The Grafcet sequence for controlling a fire damper.

have an immediate impact on the system, this operation mode has a special branch that is reached through an exception transition.

This Grafcet sequence sets logical parameters that correspond to a certain operation mode. These parameters are checked in the other Grafcet sequences when deciding what mode is active.

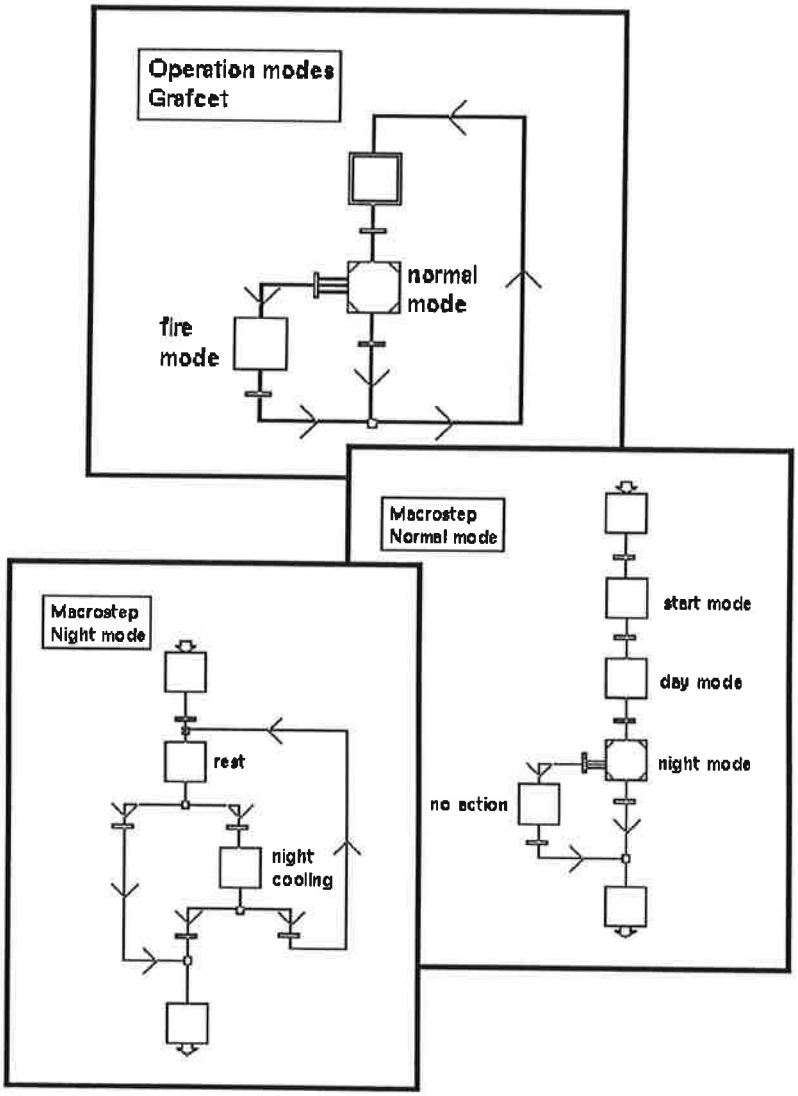


Figure 8.11 The operation mode of the system is determined with a Grafcet sequence. The fire mode must affect the system at once and is therefore governed by an exception transition from the normal mode. The normal mode is a macrostep that contains the other modes such as start, day, and night modes. The night mode is also a macro step.

9. The Control and End-User interfaces

9.1 Three user categories

A computer-based BMS has three main users. First there is a person that we call the designer. He designs and implements new function blocks and Grafcet sequences that are to be part of the library containing standard function blocks. This person should of course be authorized to use all available utilities in the program and be able to see every sublevel in hierarchical blocks etc.

The second person that uses the system is a designer that is not quite as skilled as the designer of function blocks. He works closer to the customer, perhaps in a sales office, and configures a new BMS according to the customer's demands by using predefined standard function blocks. He should also be authorized to make minor changes in the actions and transition conditions of Grafcet sequences etc.

The third one that is to use the system is the end user. This person may have some knowledge of computer-based systems but on the other hand might be a complete novice. The more skilled end user may be authorized to change some of the actions or transition conditions in a Grafcet sequence whereas the unexperienced user is only allowed to change the controller setpoints etc. Perhaps the end user has no interest in some sublevels of hierarchical blocks or macro steps in Grafcet. These should then be hidden from him so that he can only see what is happening on the superior level.

The three users might not have the same idea of what is important to see on the screen, and this we will discuss in the following section.

9.2 Different interfaces for different users

The three user categories mentioned in the previous section all have their own wishes regarding what should be displayed on the screen. The final interface thus has to be a compromise between these three points of view.

The Control interface

The purpose of the design or configuration stage is of course to create the control program of the installation. This is carried out off-line with a graphical, interactive design tool and results in the control interface. When running the system, this control interface is used on-line to debug the program or as a part of the supervision of the HVAC installation. To make the debugging and supervision a bit smoother there are some essential facilities that should be supported, and they are discussed in Section 9.3.

The off-line configuration can be carried out as follows: The person that is to

design a new BMS first decides what functions are to be part of the system – temperature control, fan control, alarm handling etc. When using the design tool he just puts these objects on the workspace area as in Figure 9.1. After choosing and placing all desired functions upon the screen, the next step will be to connect the objects to different in– and outports of the RPU. This is done on the sublevel of each object as pictured in Figure 9.1. The primary advantage with this way of creating the control program is that when you look at the screen you immediately understand what different functions the system contains. This overview is not disturbed by a lot of wires running across the picture.

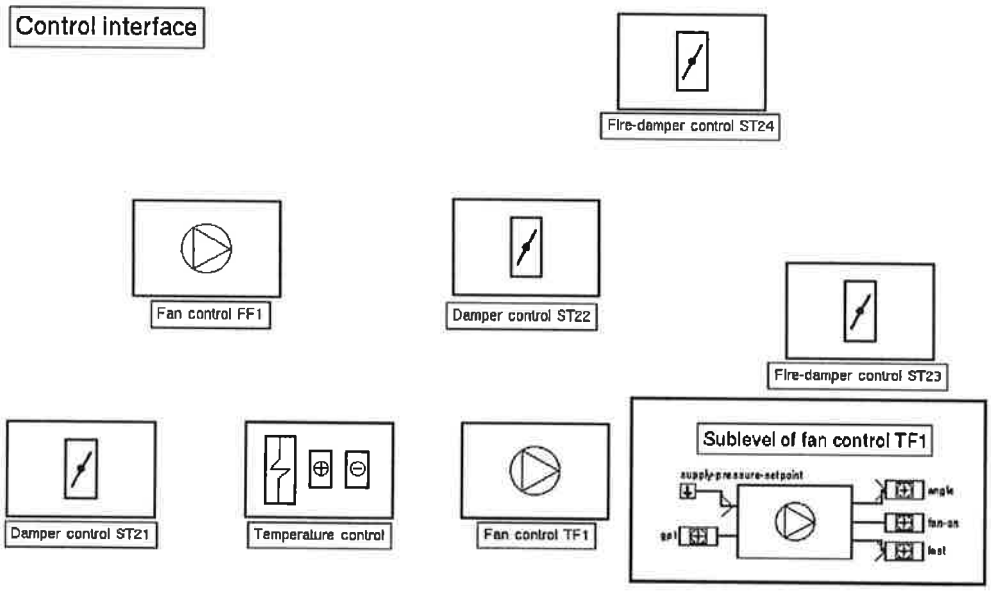


Figure 9.1 The different functions in the installation are represented by graphical function blocks that control the temperature, the pressure and the dampers. The sublevel of the block controlling the fan TF1 is shown to the right, where the small rectangular blocks represent I/O ports of the RPU.

So now the control program is ready to run, but still the end user has no access to a picture showing the status of the different physical components in the installation when it runs. This part of the system has to be created separately as an end-user interface as in Figure 9.2. This picture is in no way connected to the objects on the control interface but only shows the status of the different physical components in the HVAC installation. To investigate how the control program is performing the end user is reduced to using the control interface on-line.

Perhaps the designer of the installation instead prefers to configurate the system as in Figure 9.3. Here all the analog and digital ports of the RPU are gathered in a rack, where the objects supporting different functions are connected to the surrounding world. The difference compared to the method described above is mainly that one level of abstraction is removed. This leads to a picture that might be a bit confusing with all the wires across the workspace area. In the example of Figure 9.3 only the most important functions in the control program are accounted for. In addition there should be objects handling ,e.g., alarms, controlling pumps and valves that supply the heating coil with hot water, and this makes the picture more complex. On the other hand you get a

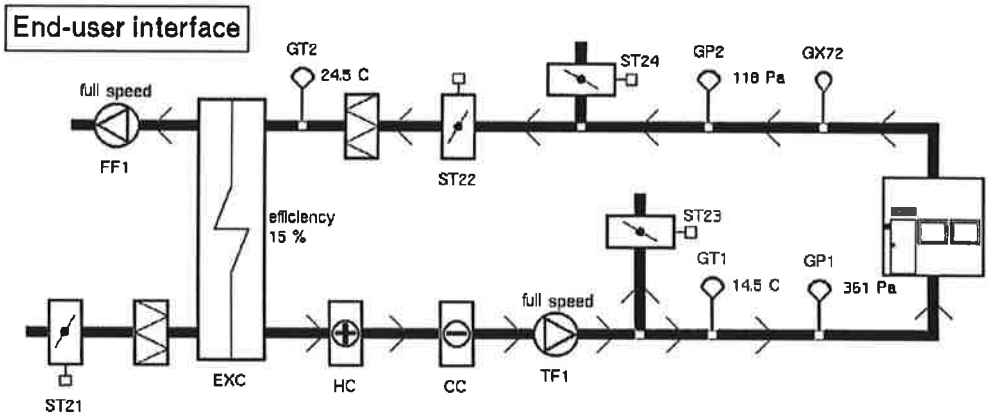


Figure 9.2 The end-user interface as seen by the person that is supervising the HVAC installation.

much better view of how the ports of the RPU are being used. The end-user interface has to be configurated separately just as before.

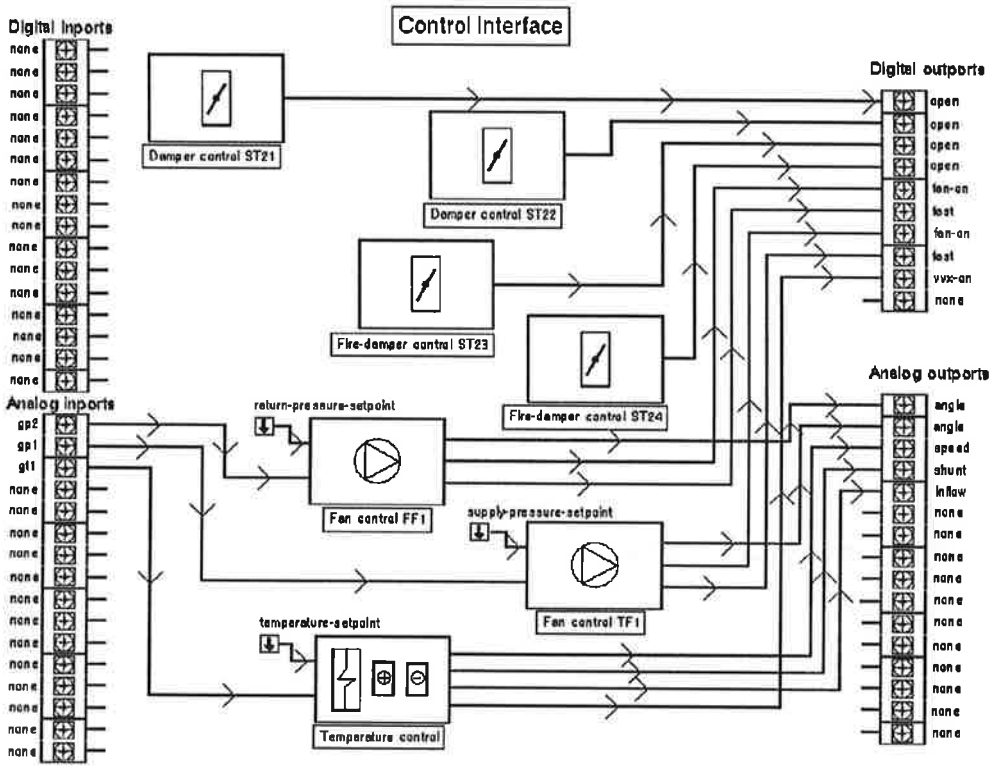


Figure 9.3 The function blocks are connected to the in– and outports of the RPU. All of the ports are gathered in a rack.

The different ways of designing a system described above require the combined structuring as described in Section 5.5. The function-based structuring is necessary when creating the control interface, and the component-based is needed when making the end-user interface.

The End-User interface

The end user is the person that supervises the air handling plant. In order to do so he needs information about the status of the different physical components in the plant, and how the control program is running. Therefore the designer must make an end-user interface in addition to the control interface. One example of such a dynamically updated interface is shown in Figure 9.2. In addition to what can be seen in this figure, an end user should have access to the setpoints and parameters of the controllers, and furthermore be able to see appropriate levels in the program, i.e. the Grafcet sequences. This must be within reach of the user from the end-user interface, e.g. by clicking with the mouse on certain buttons. Of course also the facilities described in Section 9.3 should also be available to the end user when supervising the installation. In order to give the user a better idea of how the components in the system are controlled by the program it is possible to make an interface like the one in Figure 9.4, where the objects that contain the control program of the physical components are connected to them with wires. This means that the control and end-user interfaces are merged into one. The major disadvantage is of course that the picture gets more complex and perhaps a bit confusing. The remedy for this is to hide some part of the picture from certain users. The end user should perhaps not see the function blocks that control the components in the system and they can be made transparent in this user mode. This way of combining control-system design and the operator interface, and hiding information is used in Sattline from SattControl AB.

The interface as seen in Figure 9.4 implies that the combined structuring is used, cf. Figure 5.8.

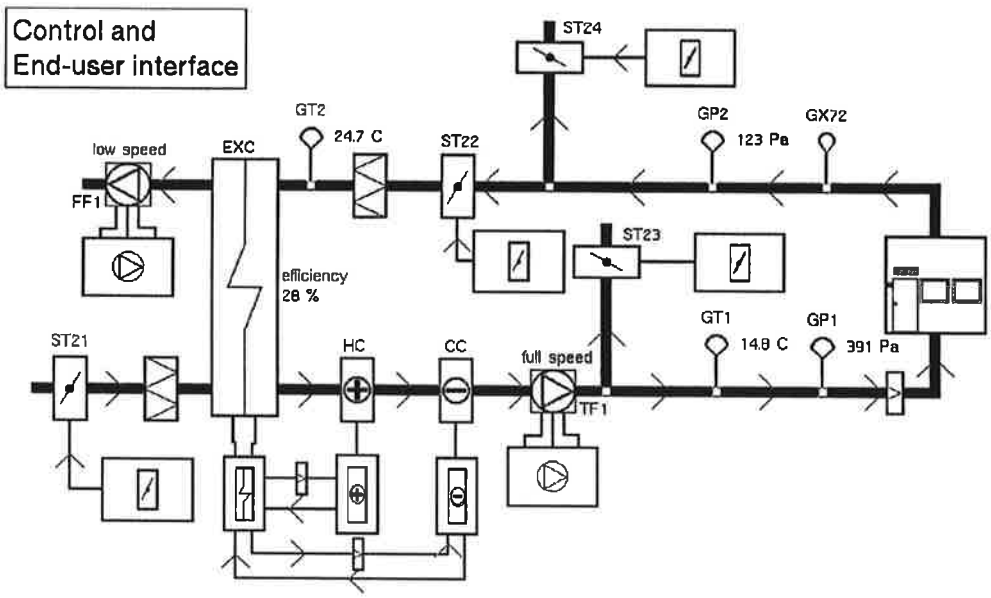


Figure 9.4 In this version of the end-user interface the objects that contain the control program are connected to the objects representing the physical components in the plant. Note that the temperature controlling object is split into three different objects, although they still communicate with each other.

9.3 Operator utilities

In order to use the control interface on-line as a debugging aid or as a tool for supervising the air handling plant some basic utilities are needed. The operator might need information about the value of attributes within the different objects on the screen. By selecting an object there should be presented a table with the attributes of the object and their values. The Grafcet sequences can be inspected by the Grafcet marker that moves through the steps, cf. Section 6.2. Finally the ability of inspecting the values on the wires is essential when debugging a program. This can be achieved by creating some sort of display at the place of the wire when selecting it through a click with the mouse. The display should be updated at certain intervals and deleted by clicking once again with the mouse on the display. An example of these facilities is shown in Figure 9.5.

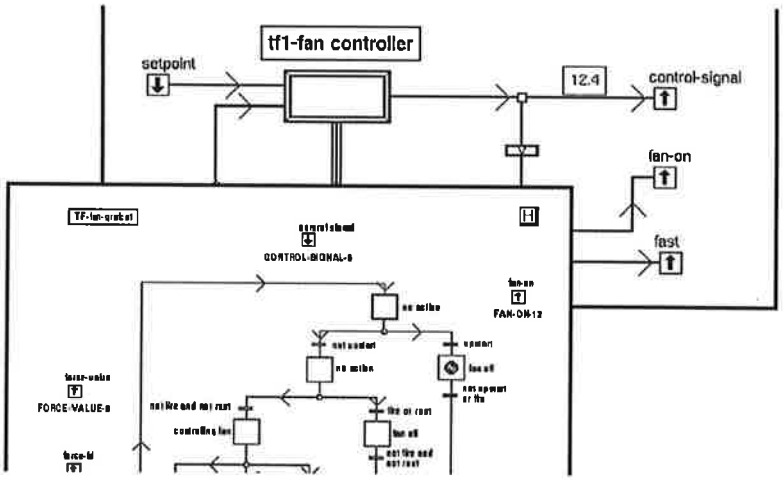


Figure 9.5 The upper part of the figure shows how the value of the wire to the control-signal is displayed. In the lower part a marker is seen in one of the steps of a Grafcet sequence.

9.4 Summary

Since we have no previous experience in working with air handling plants or building management systems it is not very easy for us to recommend one way of presenting information to the user. While implementing "our" system in G2 we have used the control interface shown in Figure 9.1, but most of the time we developed new function blocks and had no specific interest in how they were connected to ports of the RPU. So it is probably better to have a rack with the in– and outports as in Figure 9.3 when you are actually designing a new system. This implies that the end user does not have immediate access to the function blocks containing the control program of the components, but this is not too much of a disadvantage. When the end user wants to get more information about the program, one only has to make sure that an appropriate level of the object containing this part of the control program is displayed on the screen.

10. Conclusions

In this master thesis we have examined if it is possible to structure an HVAC system in an object-oriented way, thereby reducing the amount of time and money spent on the design of new systems. An additional question has been if the sequential part of the control program of such a system can be implemented using Grafcet.

By carefully studying the control program of a real HVAC system written using a line program we have found that it can very well be divided into smaller parts that control either a certain function or a specific component in the system. The system can thus be structured using at least two different approaches. Either the structure is based upon the physical components in the system or upon the control functions in the control program. Both of these approaches have appeared to be fruitful but a third way is to combine the two into one. By doing so the benefits from both the component based and the function based structuring are taken advantage of.

We found the combined structure to best serve our purposes in order to implement a graphical design tool using G2. The reason for using the combined structure is that the function based structure leads to problems when creating a dynamical end user interface for supervision of the system. The component based structure may be used but it will result in a more extensive library of standard function blocks than the combined structure. In a more extensive system than the one we have studied this will be a great disadvantage.

As for the question about using Grafcet when implementing the sequential control program we have found that this is a very powerful way of writing the program. At least the graphical picture of the program will be much easier to understand, debug and supervise. Grafcet will clearly be part of a future standard among the automation and control system suppliers.

The documentation of a graphically based control program for building management systems is one problem that has yet to be solved, since it is not possible to generate a listing of the program. One possible solution is to automatically generate a screen dump of all different graphical pictures in the system.

When working with this thesis we have found that an object-oriented structure of an HVAC system may be performed and that the sequential control program is easily implemented by means of Grafcet.

References

[1] R.L. Moore & H. Rosenof & G. Stanley. "Process control using a real time expert system," Proc. 11th IFAC World Congress, Vol 7, pp. 234–239, Tallinn, Estonia, 1990.

[2] Thomas Hartman. *Programming languages for DDC systems.* Heating/Piping/AirConditioning December 1990.

[3] Jonathan C. French. *Object-oriented programming of HVAC control devices.* ASHRAE Journal December 1991.

[4] Tour & Andersson AB, Reglerdivisionen. *Reglerhandbok–System.* 1993

[5] Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc. 1991.

[6] René David & Hassane Alla. *Petri Nets & Grafcet.* Prentice Hall 1992.

A. G2 – a tool for real-time expert systems

G2 (trademark of Gensym Corporation) is the most widely spread and technically most advanced real-time expert system tool available today. It is implemented in Common Lisp and runs on several platforms, in this project a Sun Sparcstation 2 has been used.

A.1 Technical Description

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.

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 colour, 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 class hierarchy, i.e. only single 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
- other objects

Constants can be numbers, symbolic values, logical values (i.e. true or false), and text strings. During run-time, constants can only be changed explicitly by the user. 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
- 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 may contain arbitrary values. The allowed values in a list 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 colour 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 have an iconic representation for these objects at the same time. 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. In 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.

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 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 cannot 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.

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 colour 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:

```
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 with the length of one second. 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.

Procedures

G2 contains a Pascal-style procedural programming language. Procedures are started by rule actions. 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 cannot 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.

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 or 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.

Development interface

G2 has a nice graphics-based development environment with windows (called 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 metres.

End-user interface

G2 has facilities for building end-user interfaces. Colours and animation can be used. An object icon is defined as a set of layers whose colours can be changed independently during run-time. The meta-colour 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, metres, 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.

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 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.2 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.

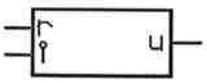
G2 can not be modularized. Hence, it requires quite powerful computers even if only a small subset of the functionality is used within an application.

Although G2 is fast, compared to many expert system tools, it can be too slow for certain applications. The smallest time unit is one second. For applications that require faster response, G2 is inadequate.

B. Function block implementation

B.1 Input, Execute, and Output Procedures

As briefly mentioned in Chapter 6 we must make sure that there is a flow of information between the function blocks. We have achieved this by having three different procedures in every object. The input procedure reads data from the inputs, the execute procedure processes data, and the output procedure writes the result of the computation on the outputs. In the root class that is superior to all other classes three empty default procedures are declared that do nothing. Every subclass has its own input, execute and output procedures and thus they are redefined in the separate classes. Should we, however, have the desire to do nothing we can let the inherited procedure be left as it is.




```
input-controller(b: class super-block)
begin
  conclude that the setpoint of b =the val of the super-block-
    connection connected at the inport1 of b;
  conclude that the measured-value of b =the val of the super-block-
    connection connected at the inport2 of b
end
```

```
exec-pid(b:class super-block)
e,y,du,p,i,d: quantity;
begin
    y = the measured-value of b;
    e = y - the setpoint of b;
     p = the k of b * (e - the e1 of b);
      i = the k of b/the ti of b * e;
      d = the k of b*the td of b*(y - 2 * the y1 of b + the
        y2 of b);
     du = p + i + d;
    if the out of b = the umax of b and du > 0 then
      du = 0;
     if the out of b = the umin of b and du < 0 then
      du = 0;
    conclude that the u of b = the u of b + du;
    conclude that the y2 of b = the y1 of b;
    conclude that the y1 of b = y;
    conclude that the e1 of b = e;
    if the u of b >= the umax of b then conclude that the
      out of b = the umax of b
    else if the u of b <= the umin of b then conclude that
      the out of b = the umin of b
    else
    conclude that the out of b = the u of b;
end
```

```
output-controller(b:class super-block)
begin
  conclude that the val of the super-block-connection connected at the
    outport of b = the out of b;
end
```

Figure B.1 This is an example of input, execute and output procedures for a PID controller written in G2 syntax.

One example of input, execute and output procedures is shown in Figure B.1. The procedures are written using G2 syntax, but we do not think this will make the understanding more difficult since it resembles, e.g., Pascal. The input procedure of every object fetches data from the incoming wire and puts

it in the attributes corresponding to the inputs. In Figure B.1 the procedure reads the setpoint and feedback input from the inputs. The execute procedure processes the data, and in this example it computes the output signal from a PID controller. Finally, the output signals are written on the outgoing wires and in this case we only have the output signal from the controller to take care of.

B.2 The execution order for function blocks

Initially when the program is started the function blocks are sorted in execution order. The function blocks are stored in a list, the execution list. The elements in this list are executed cyclically and the execution of a block involves the execution of the input, execute, and output procedures in that order. Blocks that need no input data in order to produce output data, such as constants or inports, are placed first in the list. All remaining blocks need data from function blocks that are connected to their inputs. When all of the blocks connected to the inputs of a certain function block have been put into the list this block can also be placed there. A problem arises when algebraic loops occur, e.g. as in Figure 7.3. The solution is to break the loop with a special block that is sorted into the list together with the blocks that need no input data.

The hierarchical function blocks all have their own execution lists since the blocks on the sublevel must be executed before the execution on the superior level can proceed. The execute procedure of a hierarchical block thus goes through the execution list of the block and executes the function blocks in it. In this case the connection posts that transfer data to the sublevel are sorted into the execution list before any other block.

C. Simulation equations

In order to verify that our control program created in G2 behaves satisfactory we have simulated the air-handling unit. The physical items, as e.g. the heat exchanger and the heating and cooling coils, all have been simulated using simple first order transfer functions provided by TA. The simulation could of course have been improved using higher order transfer functions but in our case the quantitative result was not the main interest. Concerning the notation in the simulation equations cf. Figure C.1.

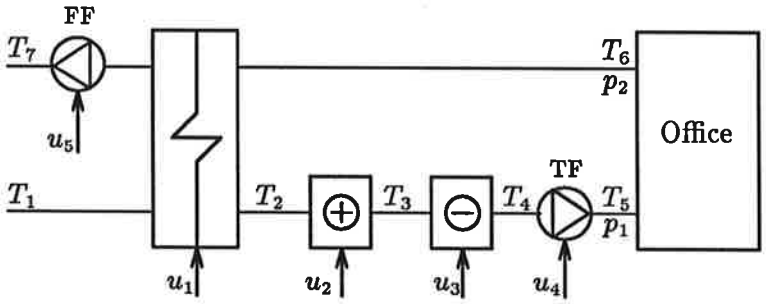


Figure C.1 The notation in this appendix is in accordance with this picture.

Heat exchanger

$$T_2 = \frac{1}{1+sT}(T_1 + \frac{\eta_0}{10}\sqrt{u_1}(T_6 - T_1))$$

$$T_7 = \frac{1}{1+sT}(T_6 - \frac{\eta_0}{10}\sqrt{u_1}(T_6 - T_1))$$

with nominal efficiency $\eta_0 = 0.75$, time constant $T = 15$ s. and control signal $0 < u_1 < 100$

Heating coil

$$T_3 = T_2 + \Delta T_2$$

$$\Delta T_2 = 10\frac{K(T_v)}{1+sT}\sqrt{u_2}$$

$$K(T_v) = \begin{cases} 0 & ;T_v < T_2 \\ 0.004T_v & ;T_v \geq T_2 \end{cases}$$

with $T = 30$ s., $0 < u_2 < 100$ and the hot water temperature T_v depending on T_1 according to Figure C.2

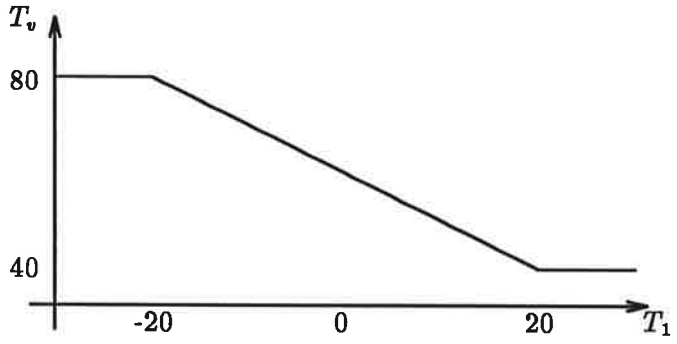


Figure C.2 The hot water temperature T_v depends on T_1.

Cooling coil

$$T_4 = T_3 - \Delta T_3$$
$$\Delta T_3 = \frac{0.12}{1 + sT} u_3$$

with $T = 30$ s. and $0 < u_3 < 100$

Fans

The temperature was simulated according to:

$$T_5 = \frac{1}{1 + sT}(T_4 + \Delta T_4)$$

with $\Delta T_4 = 1$ (slow) or 2 (fast) and time constant $T = 5$ s.
The pressure was simulated according to:

$$p_1 = \frac{k_{TF}}{1 + sT} u_4$$

$$p_2 = \frac{k_{FF}}{1 + sT} u_5$$

where $k_{TF}, k_{FF} = 3$ (half speed) or 5 (full speed), $T = 30$ s. and $0 < u_4, u_5 < 100$

Office building

$$T_6 = \frac{1}{1 + sT}(k_5(T_5 - T_o) + T_o)$$

where $k_5 = 0.4$, $T = 30$ min. and $T_o = 25$

A simulation session

During a simulation in G2 the end user interface is dynamically updated with respect to sensor temperatures, sensor pressures, efficiency of the heat exchanger, fan speed, etc. There is also a possibility of showing a plot of the sensor temperature and sensor pressure in a graph. In Figure C.3 an example of a simulation in G2 is shown.

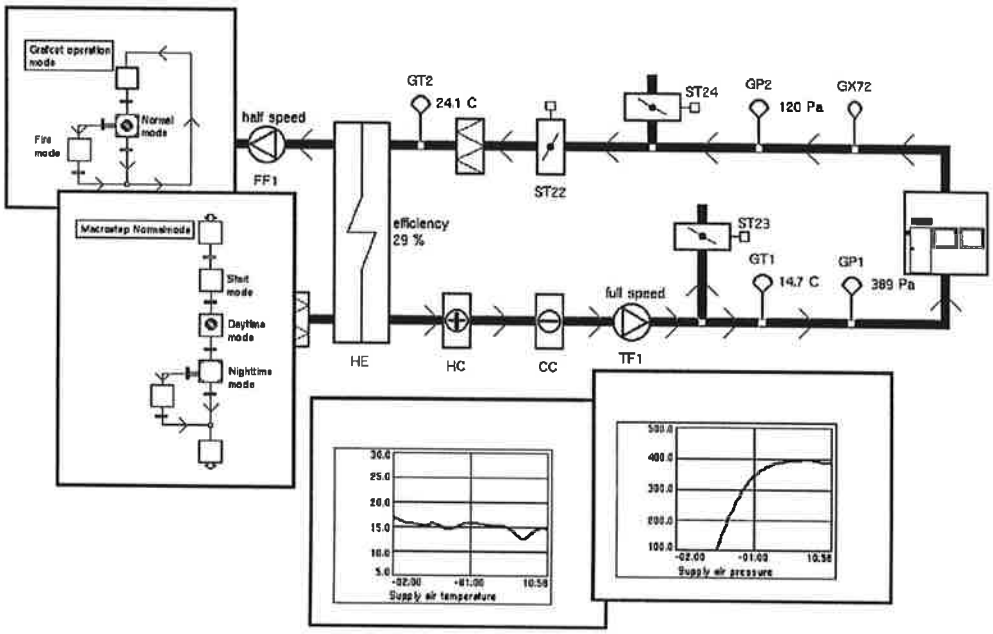


Figure C.3 A simulation of the HVAC system in G2. The Grafcet sequences and the graphs can be moved around on the screen by the user.