Object-Oriented Programming of PLC based on IEC1131

Hartmut Weule, Dieter Spath, Hans-Joachim Schelberg

Abstract

The software development for programmable logical controllers is usually based on low-level languages such as the instruction list or the ladder diagram. At the same time, the programmer looks at a machine or an assembly system in a bit-oriented way: he translates the operational sequences into logical and/or time based combinations of binary signals described by the means of boolean algebra. A machine, however, does not only consist of binary signals but of technical components, i.e. objects such as nc-axes, carriage systems and other facilities. These objects can be characterized by different features; their connection to a PLC being the basis of the conventional programming methods is only one of them. As these classical methods cause a lot of problems in reality they should be improved. The following report shows a way how to integrate object-oriented features into the standardized PLC-language "Sequential Function Chart".

Keywords: *Computer Aided Engineering, Object-Oriented Programming, Programmable Logical Controllers*

Introduction

Most of the faults that are being detected during the putting into operation of complex production facilities are caused in earlier phases: in development, in design, in software engineering, in assembly. Especially, the point "Software engineering" seems to be critical: different investigations have shown that about 70-80% of all problems during the first operations can be traced back directly or indirectly to software bugs. Analysing this situation one can find different aspects, as fig. 1 demonstrates.

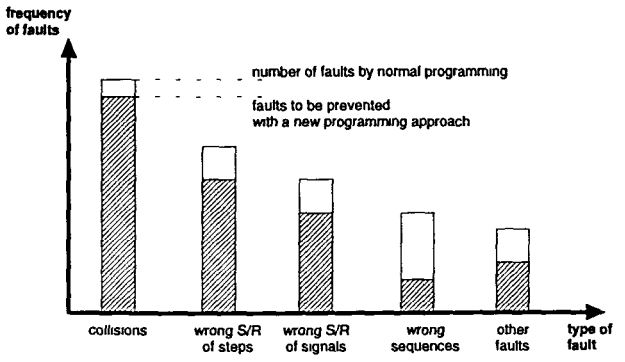


Fig. 1: Software-Faults in first operation phases of automated production facilities

On the one hand we can find systematic errors like wrong sequences. They are generally the consequence of the technical documents given to the software engineer. These instructions are created by the designer who generates the mechanical functions and operational sequences of the machine. As they are the only working basis for the programmer (who has to carry out the functions by software) they should include at least the geometric layout of the machine, a list of the active and passive technical components with their connections to the PLC and a complete description of the functions the machine has to perform. So far with theory, in reality the papers are often incomplete, inconsistent or sometimes simply wrong. Therefore, there is no doubt that most of the errors they cause could be avoided by organizational means [1,2].

On the other hand technical errors can be distinguished. Wrong setting and resetting of program steps or signals (with the - in most cases - harmless effect that nothing happens; the machine comes to a standstill) are typical effects of the hardware-based, connection-oriented languages and the tools used for software development. They force a more or less bit-oriented view on the process: the state of an object or a complex event is reduced to logical and/or time based combinations of binary signals described by the means of boolean algebra. In the same way, actions or movements of an object are set off by setting and resetting other binary values. The assignment between the states of different signals to a certain movement is not recognizable directly but in the source code comments (as far as there are some) or technical descriptions. So there is no way to detect logical faults in the software automatically by engineering tools because of the very simple semantics of the common lanuguages. An exceptional position take the collisions which generally happen among different active elements or among active elements and workpieces; further to the before mentioned reasons they happen due to the increasing complexity in case of parallel or time-optimized sequences, which are very difficult to handle.

To solve most of these problems the classical method of programming isn't efficient any more as seen before. A new point of view to a machine, which also should have consequences for the programming languages and the development tools, is necessary. Fig. 1 gives a hint of the possible fault reduction when performing the change in a consequent way. But first, what are the boundary conditions and constraints for a new approach?

Boundary conditions

The new standard for PLC-programming languages is the IEC1131. In order to support the portability of PLC-programs among others the basic languages "instruction list (IL)" and "sequential function chart (SFC)" are defined by their syntactic and semantic charactaristics [3].

On the condition that a new programming method should be compatible to these standardized ones the SFC seems to be an appropriate language to build on. Especially, in case of assembly systems with their generally sequential, event-based charactaristic the SFC is a very useful method to describe the process. Furthermore, it has some other advantages; representing a special kind of place-transition nets it is derived from the petri-net theory. Therefore, the mathematic algorithms to proove reachability or deadlock freeness etc. can be applied easily to analyse the source code before testing the program with the real machine [4].

The steps and transitions of the SFC are to be programmed by other languages, e.g. the instruction list (IL), which are "responsible" for the technical errors described in the beginning. The IL is a typical bit-oriented language: orders consist of a logical expression combined with the bit-adress of a process element. So, there is a starting point for a new programming approach: in the Microcomputer-world we can ascertain an actual trend to object-oriented languages like EIFFEL or C++ because of their great advantages in software extensibiltiy and reusability. At the same time, they are quite a revolution for the design of software systems: in contrast to the function-oriented methods of the past now the objects with their behavior and their interdependences between one another are the basis of programming. As a production system can be structured into different objects, e.g. as proposed in VDI3260 [5], it is obvious that this method can be adapted for PLC-programming, too.

Defintion of basic object classes

Now, the basic objects to be considered for PLC-software are the so-called "functional units" like sensors, pneumatic cylinders or nc-axes. These technical components of a machine feature different aspects, as shown in Fig. 2.

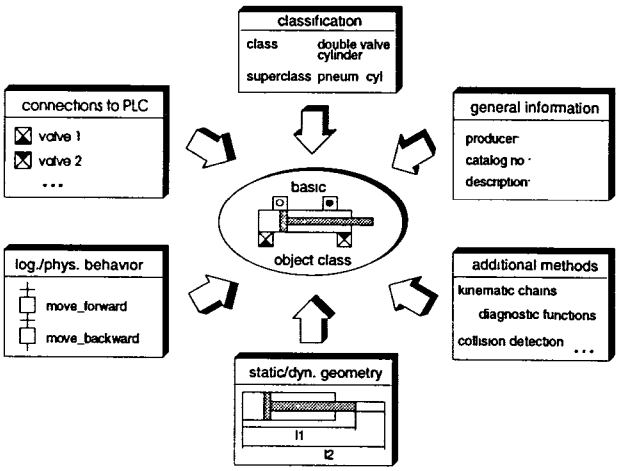


Fig. 2: Definition of basic object classes

In order to create freely extendable classified libraries of objects the first attribute is of course the name of the object class, e.g. "double valve cylinder" and the superclass it is derived from. Due to this dependency common characteristics of different cylinders like their logical behavior can be generalized. The other way round, an object inherits attributes and methods from its superclasses (multiple inheritance). This mechanism involves a lot of consequences for programming as to be seen later.

The next important attributes are the number and the names of active and passive connections to the PLC. Less for programming but for the final PLC-code generation we have to describe in which way an object is connected to the PLC.

The most important feature of an object is its logical and physical behavior, where the two aspects "object's logical reactions on PLC-orders" and "logical feedback of its current state" are the basic ones. The easiest way to model this feature is to use the same SFC as before, although some other forms as the state diagram are possible as well. Fig. 3 shows the behavior of a simple pneumatic cylinder, which is connected to the PLC by two active valves (A1, A2) and two passive sensors (S1, S2). It has four states which we can also call (in analogy to the object-oriented modelling technique [6]) "methods": the piston is in forward or backward movement ("move_forward", "move_backward") or it is in front or back position ("is_infront", "is_back"). Allthough we have in fact 'static' (which return the actual state of an object) and 'dynamic' (timebased actions) methods we are treating them the same way. The names of these methods can be chosen freely by the programmer or by the enterprise, but the creation of standard libraries for modular construction systems is conceivable, too.

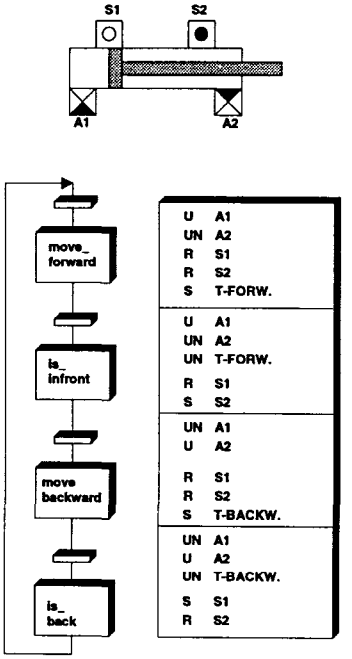


Fig. 3: Logical behavior of a simple pneumatic cylinder

From the view of the cylinder the conditions of each method are definite. If the piston shall move forward, the PLC always has to set A1 and to reset A2. Yet, the sequence of the methods is always the same: usually it does not make any sense for this cylinder to move back if it is just in forward movement. This allows an extended analysis of the source code to find out wrong sequences of orders or orders excluding one another.

The other features shown in fig. 2, i.e. the "general information", the "static/dynamic geometry" and the "additional methods", can be omitted at this stage, although especially the latter gaine high importance in a current extension of the reports subject. So, the basic information for programming the objects directly is complete. The next step is to define a "language" supporting the identification of certain objects and their actions.

A new Object-oriented programming language

A complex machine consists of lots of basic objects like the pneumatic cylinder described. Therefore, we have to identify every single unit by a unique attribute. The classical way of identification is the symbol list containing only the physical connections to the objects. We can call the equivalent form "object list" (fig. 4), where every object to be controlled in the machine is clearly charactarized by its instance-name and class.

project:	*pressing system*	date: 17.03.93	
object	class	connection	parameter ...
pressure_ram	*double valve*	S1 = PCYL_IB	T-FORW. = 3.0 sec
	cylinder	S2 = PCYL_IF	T-BACKW. = 1.0 sec
		A1 = PCYL_MF	
		A2 = PCYL_MB	
ejector	*single valve*	S1 = EJECT_IB	T-FORW. = 1.0 sec
	cylinder	S2 = EJECT_IF	T-BACKW. = 0.5 sec
		A1 = EJECT	
⋮	⋮	⋮	⋮

Fig. 4: Section of an object list

To define the "language" we make use of these names and the methods belonging to the object classes. If a certain cylinder shall move forward (action) we simply write:

pressure_ram (object) *move_forward* (method)

The same we do with transitions:

U/O/UN...(log. cond.) *ejector* (object) *is_back* (method)

The introductory logical expression in the transition is important when there are multiple conditions; in actions there is an implicit 'Set', which we do not have to write down explicitly. So, in that way we can look at this new 'language' as an equivalent or a modified form of the usual instruction list (the IEC-standard is quite flexible at this aspect), where the symbols and the orders are replaced by objects and their methods.

Now, what are the effects of this way of programming? A little example (fig. 5) shall demonstrate them:

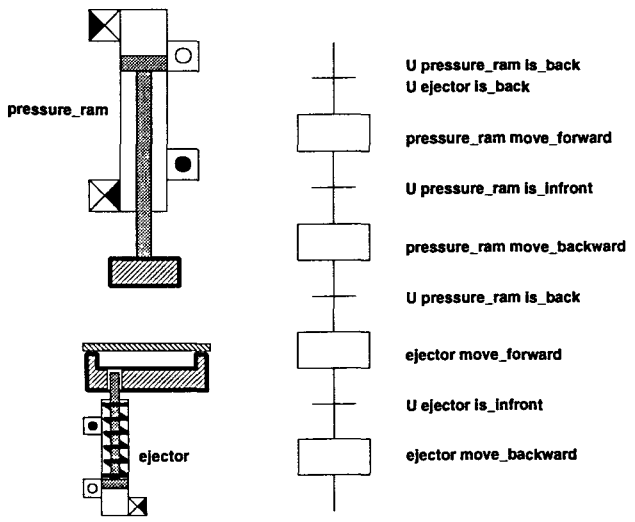


Fig. 5: Example of an Object-Oriented-SFC

First, we have lifted up the programmer's view to a machine from the "connection layer" to a more abstract "object layer": he doesn't have to know any more which combinations of which binary signals cause e.g. a certain cylinder to move forward. He can simply pick the desired objects from a library, name them, fill their parameters and then use their methods. Of course it can be argued "This would be possible with subroutines, too!", but it makes a great difference, whether one has to write down one simple word only or to pass the right parameters to the right subroutine. Moreover, this would lead to a huge number of subroutines. This can be avoided by an object-oriented approach as fig. 5 shows: different cylinders inherit the same methods from their basic classes. The translation of these methods within the "Object-Oriented - SFC", using the object list and the class definitions, is now the task of the compiler, not that of the programmer. In other words, the new language supports polymorphism. Furthermore, to guarantee full compatibility to the language standard IEC1131, the output of the compiler's code generator is in the first pass the same SFC combined with a "normal" instruction list.

So, the object-oriented SFC shows a lot of advantages: The programs become shorter and are much easier to create. Anyone can understand and check such a program sequence at a glance. This is especially important for designers and for service personnel. At last, this method supports the avoidance of the technical errors mentioned in the beginning of this report. Wrong setting and resetting of signals being reasons for wrong sequences are quite impossible with this method.

Collision detection

Collisions are the most ugly faults occuring in the (first) operation phases of an automated technical system. They lead to high costs in replacing defect parts, deficits in production capacities and output etc.. Potential reasons for them can be found in different aspects:

Machine

- wrong connections between controllers and objects
- mechanical, pneumatical, hydraulical or electrical malfunctions
- wrong workpieces or tools in process

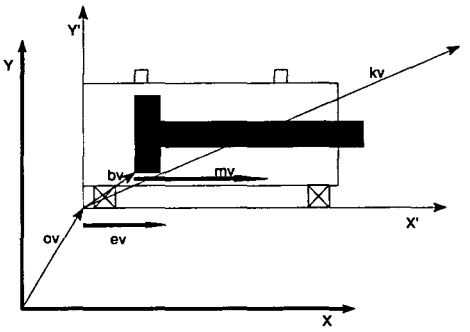
Software

- programming bugs as described in the beginning
- missing locks between components
- wrong coordination of movements because of a combinatorical explosion of possible states
- insufficient tools and methods

As we do not have any influence on the mechanical reasons in our case we aim at the software aspects now. But how can we detect possible collisions?

Beyond the logical model which is the basis of the new programming language described before we need a geometrical and dynamical model of a technical component. This model should represent the temporal (time profile of the active components, physical behaviour) and two- or three-dimensional movements (possible positions of the active components) of an object. Fig. 6 shows the two-dimensional geometric description. First we can find a vector to the basic position of an object in the whole system. The next vector decribes the orientation of the object in the machines co-ordinate system. If there are several

objects combined to a complex component, i.e. a robot, we can describe the dependences between its basic objects by the chaining vector. With the aim to detect collisions however the most important vectors are the one to the dynamical component(s) of an object and their movement vectors. Completed by the speed profile of the movement we now have the basis for this task.



ov = vector to basic position of the object
ev =onentation of the static part
kv = chaining vector to another object
bv = vector to a dynamical component
mv = movement vector of the dynamical component

Fig. 6: Simple geometrical and dynamical model of an object

The next step is to examine the possible interactions between two objects. With the help of a three pass test shown in fig. 7 we got a quite simple and very fast technique to find out if two or more objects can collide with each other. Furthermore, we can find out the logical and time-based conditions for critical situations.

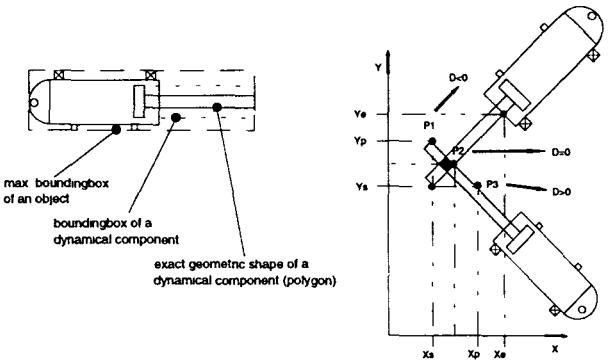


Fig. 7: Mathematical test to detect intersections between objects

In the first pass we are testing the maximum size of all objects by intersecting their maximum bounding rectangles. The same we do with the bounding boxes of the dynamical components. If there are still possible colliisions, we calculate them exactly by intersecting their polygons. The solution of the polynom describing the dependences between the objects indicates at which time and position a collsion could happen. The results of these tests can be handled then by the compiler which translates the new programming language.

So, with this technique we can find out any possible collisions between objects automatically. The basis for this method is of course to create a computer based model of the system to be controlled. Such a tool called "TOPAS" has been realized since 1991 at the wbk [7]. A special feature of it allows to automatically prevent any collisions by generating code-sequences out of the machine model. Inspite of the effort to create this model the consequences of this approach are highly promising: Even if we can not find out all possible faults in the machine, we can eleminate most of the faults in the first operation phases before testing the code with the real technical system. Therefore this phases will be shorter, cheaper and with less risk in future. The model itself can be used for different tasks then: to create further models, for process visualization, for online-simulation or simply for training the employees working with the machine later.

The consequent extension of this approach is the connection to a system, which automatically creates a way to solve the problems if the machine has failed [8]. For this purpose it avails on a similar logical model of the machine`s functional components. So, the input information for that system can be generated by the technique described in this report in a comfortable manner.

Summary

Software development for PLCs is a difficult and troublesome work on the basis of classical programming methods and languages. Software bugs are very nasty, especially in process-controlling software as in our case. The object-oriented approach described in this report may be a way to improve and to ease software development in production automation.

In this way the functional objects of a machine become more intelligent: supported by new technical means like distributed controllers, self-controlling pneumatic elements they simply ask the objects in their surrounding "May I ..." before doing anything.

References

1. *Janzen, F.:* Projektierung von SPS in einer integrierten, rechnergestützten Automatisierungsumgebung. Diss., Ruhruniv. Bochum 1990
2. *Weck, M; Kohring, A., Aßmann, S.:* Bereichsübergreifende Spezifikation von Werkzeugmaschinen. VDI-Z 5/92, S.111 - 115
3 *Trunz,W, Bender, K.:* Ein Sprachstandard zur herstellerunabhängigen Programmierung von SPS-Systemen. atp 3/91, S. 128 -137
4. *Lenschow, R:* Rechnerintegrierte Erstellung und Verifikation von Steueungsprogrammen. Dissertation, Univ. (TH) Karlsruhe 1991
5. *VDI3260, Abschn. 2:* Funktionsdiagramme von Arbeitsmaschinen und Fertigungsanlagen: Aufbau der Fertigungsanlage. Berlin, Köln: Beuth Verlag 1977
6. *Rumbaugh, J. e.a:* Object-Oriented Modeling and Design. Prentice-Hall International, 1991
7. *Schmidt, J.; Schelberg, H.-J.:* Objektorientierte Projektierung von Steuerungssoftware. VDI-Z 12/91, S 60-65
8. Enderle, W.. Verfügbarkeitssteigerung automatisierter Montagesysteme durch selbsttätige Fehlerbehebung. Dissertation, Univ. (TH) Karlsruhe 1989